DISPLAYING PARAMETRIC CURVES AND SURFACES USING UNIGRAFIX

DISPLAYING PARAMETRIC CURVES AND SURFACES

USING

BERKELEY UNIGRAFIX

By

HENRY CHEUNG, B.Sc.

A Report

Submitted to the School of Graduate Studies

in Partial Fulfilment of the Requirements

for the Degree

Master of Science

McMaster University

(c) Copyright by Henry Cheung, August 1988

MASTER OF SCIENCE (1988)
(Computation)

McMASTER UNIVERSITY
Hamilton, Ontario

TITLE: Displaying parametric curves and surfaces using Berkeley UNIGRAFIX

AUTHOR: Henry Cheung, B.Sc.Hons. (Queen's University)

SUPERVISOR: Professor Patrick J. Ryan

NUMBER OF PAGES: vii, 180

ABSTRACT

Henry Cheung: Displaying parametric curves and surfaces using Berkeley UNIGRAFIX. M.Sc. report, McMaster University at Hamilton, August, 1988.

Berkeley UNIGRAFIX is a graphics system that runs under the UNIX operating system. It comprises a collection of rendering programs and scene generating programs. Scenes of objects are described in a terse, human-readable format called the UNIGRAFIX descriptive language.

In order to display parametric curves and surfaces using UNIGRAFIX, we developed a scene generator called UGTRACE. It traces a set of parametric equations and generates a scene file written in the UNIGRAFIX format. Two tracing modes as well as two displaying modes were designed.

The Berkeley UNIGRAFIX system was installed in the McMaster environment. For our purpose, we also modified the various device drivers of UGDISP, the latest version of the renderers. Available output devices include the AED colour graphics terminal, the IMAGEN laser printer and the IBM PC emulating a Tektronix 4010 terminal. UGDISP is used to display scene files generated by UGTRACE.

ACKNOWLEDGEMENTS

It is a pleasure to acknowledge my gratitude to my project supervisor, Professor Patrick J. Ryan of the Department of Computer Science and Systems. Professor Ryan suggested the problem, and his encouragement and friendly guidance in the ways of the project made the work a rare educational experience.

I would also like to express my appreciation to my wife Betty for unflagging patience and support.

And above all, to God, from whom all good and perfect gifts come, be my thanksgiving.

CONTENTS

1. INTRODUCTION

1.1 The Berkeley UNIGRAFIX system 1
1.2 UGTRACE: a UNIGRAFIX generator for parametric curves and surfaces 4
1.3 Project profile 7

2. DESIGN

2.1 Device independent plotting 10
2.2 The abstract command set 12
2.3 The AED 512 colour graphics terminal 13
2.4 The IMAGEN 3320 laser printer 17
2.5 The TEK 4010 terminal 18

3. UGTRACE: A GENERATOR

3.1 Overview 21
3.2 Input section 22
3.3 Tracing section 33
3.4 Output section 36

4. SUMMARY 41

BIBLIOGRAPHY 44

Appendix A. USER'S MANUAL

A.1 ugdisp 46
A.2 ugtrace 52

Appendix B. TUTORIAL 54

Appendix C. EXAMPLES 61

Appendix D. ALGORITHMS AND DATA STRUCTURES FOR UGTRACE

D.1 Overview
D.1.1 Control flow 72
D.1.2 Source files 73
D.2 Input section
D.2.1 Scanning command line [phase 1] 74
D.2.2 Loading specification [phase 2] 76
D.2.3 Parsing specification [phase 3] 76
D.2.3.1 Parsing Shell 76
D.2.3.1.1 Parsing a CONSTANT statement 77
D.2.3.1.2 Parsing a PARAMETER statement 78
D.2.3.1.3 Parsing a VARIABLE statement 79
D.2.3.1.4 Parsing a DOMAIN statement 81
D.2.3.1.5 Parsing an equation 82
D.2.3.1.6 Parsing an expression 83
D.2.3.2 Overall checking 88
D.3 Tracing section
D.3.1 Tracing [phase 4] 89
D.3.1.1 Unit-step tracing 91
D.3.1.2 Unit-length tracing 93
D.3.1.3 Expression evaluation 95
D.4 Output section
D.4.1 Exporting to UNIGRAFIX format [phase 5] 95
D.5 Function directory 98

Appendix E. SOURCE CODE FOR UGTRACE 100

FIGURES

1.1 Displaying objects using Berkeley UNIGRAFIX 2

1.2 Illustration of tracing modes using a sine curve 6

2.1 UNIGRAFIX output interface 11

2.2 UNIGRAFIX plotting modes 15

2.3 PC/UNIGRAFIX interface 19

3.1 Command line switches 23

3.2 Syntax of input text 25

3.3 Grammar used for expression 30

3.4 An example of UGTRACE internal representation, tracing section 32

3.5 Scenario for unit length tracing of parametric surfaces . 37

3.6 UGTRACE internal representation, output section 37

3.7 UNIGRAFIX syntax, abridged summary 39

3.8 UGTRACE display modes 40

C.1 Unit-step tracing versus unit-length tracing 63

C.2 Varying tracing magnitude and using multiple domains . . 67

C.3 Mesh display versus Patch display 70

Chapter 1

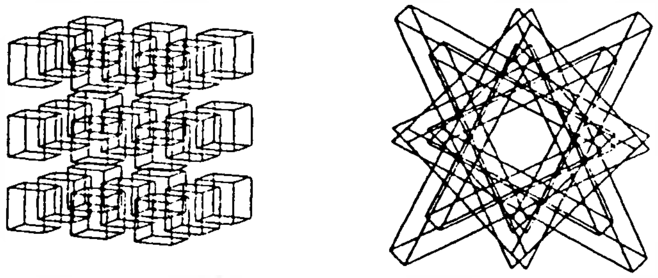
INTRODUCTION

1.1 The Berkeley UNIGRAFIX system

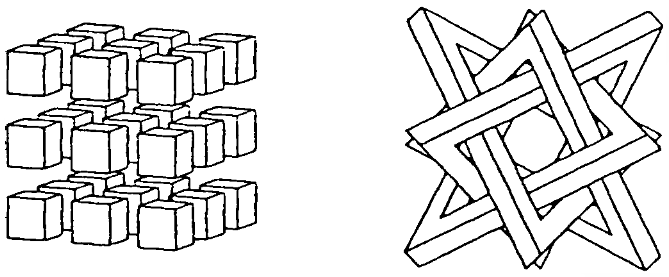
The UNIGRAFIX system [SEQU83a-c] is a batch-oriented graphics system running under the UNIX operating system [1]. It was started in 1981 at the University of California, Berkeley campus [2]. Since then it has been a constant focus for master's theses and course projects. The purpose for developing such a system is to provide some rendering tools for geometric modeling under UNIX. The desired goal is to produce scenes of two or three dimensional polyhedral objects in the style of engineering drawings. Output is primarily a high resolution, black and white, dot raster.

The current system comprises several renderers, and a collection of object generators and modifiers. Object scenes can be rendered, basically, in two different ways. Under the wire-frame format, skeletons of the objects are shown by displaying all the edges, giving an X-ray view of the scene (Figure 1.1(a)). And for a more natural appearance, scenes can be displayed with hidden

[1] UNIX is a trademark of Bell Laboratories.
[2] The U. C. Berkeley UNIGRAFIX system has been supported by Tektronix, Inc. and by the Semiconductor Research Corporation.



(a) Wire frame format



(b) Hidden parts and overlappings removed

Figure 1.1 Displaying objects using Berkeley UNIGRAFIX

edges and surfaces removed (Figure 1.1(b)). Light sources are used to give shades, providing an artistic finish to the display. In addition to the displaying modes, viewing parameters such as eye point coordinates and viewing direction can also be adjusted at run-time. The renderers support a wide range of plotters and graphics terminals.

The generators provide the means to design objects such as mechanical parts [3] or architectural elements [4], as well as purely geometrical objects in three and four dimensions. The modifiers provide the means to manipulate simple objects. Complex scenes are constructed from simple primitive objects either by truncating them, by tessellating their faces, or by cutting holes into them. In the fields of robotics and computer vision, some of these utilities may be found useful.

Since UNIGRAFIX is designed to be developed over a period of time and contributed to by different people at different stages, it is important to maintain the coherency and the consistency of individual work. For this purpose, the UNIGRAFIX descriptive language was developed. The language is a powerful yet simple descriptive format in ASCII text used to describe an object or a scene. All modules of the UNIGRAFIX system use this as a common format for specifying scenes. Thus the UNIGRAFIX language provides a standard interface between modules which glues

[3] e.g. gear wheels and robot arms
[4] e.g. staircases and houses

the whole system together. As a result, each module is allowed to be developed independently and to be replaced or upgraded when newer and better ones are available. For technical details of UNIGRAFIX, see the publications listed in the BIBLIOGRAPHY section.

1.2 UGTRACE: a UNGRAFIX generator for parametric curves and surfaces

The prime objective of developing UGTRACE is to provide a tool for displaying mathematical curves and surfaces given in parametric form. In keeping with the UNIGRAFIX design philosophy, this is done by adding a new generator UGTRACE. For parametric curves in three dimensional space, the format in general is as follows:

$$\begin{aligned} x &= f(u) \\ y &= g(u) \\ z &= h(u) \end{aligned}$$

where u ranges over some parameter domains. If the variable z is dropped, then the curve is confined to two dimensional space only. On the other hand, surfaces require two parameters:

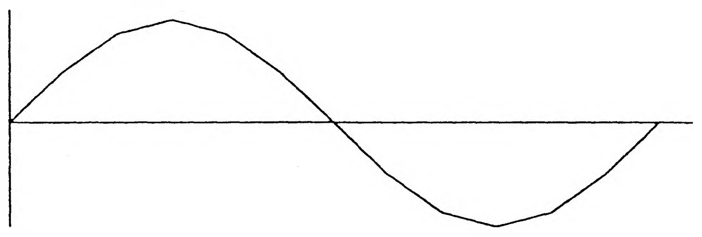
$$\begin{aligned} x &= f(u,v) \\ y &= g(u,v) \\ z &= h(u,v) \end{aligned}$$

For more background on parametric curves and surfaces, see Foley and Van Dam pp.514 and/or any multivariable calculus text.

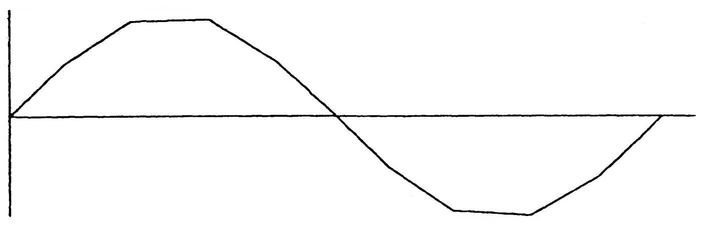
Given a set of parametric equations and their parameter domains, the program computes a representative set of vertices, edges, and faces, and then produces a description of this scene in the UNIGRAFIX language. Although this script in the UNIGRAFIX language is readable, it is basically a list of numbers and hardly gives the impression of a pictorial scene. It must be fed into one of the UNIGRAFIX renderers before the scene can be visualized.

An infinite number of points exists on any curve or surface theoretically. Tracing each and every one of them would be impossible. In practice, a sampling of points must be chosen. UGTRACE is capable of sampling points using two different algorithms. One method is called unit-step tracing. Parameters are changed at an uniform step size to get the points. Depending on the characteristics of the functions, distances between successive points may not be uniform (Figure 1.2(a)).

Another method used is called unit-length tracing. Under this method, the distance between successive points is required to be the same for all intervals. Instead of keeping the step size unchanged, trial sizes are used and adjusted repeatedly until the distance equals the required length. In the end, a set of equally spaced points is gathered (Figure 1.2(b)). This method, in comparison to unit-step tracing, requires more computing effort.



(a) Unit step tracing, parameter step size at PI/6



(b) Unit length tracing, points equally spaced at 0.75 unit of length

Figure 1.2 Illustration of tracing modes using a sine curve whose equations are x = t and y = sin(t).

The set of points generated using either method gives an approximate representation of the curve or surface. The tracing process is iterative. If the approximation is considered not satisfactory, then the user can always change the step size or length magnitude and run the program again. However, there is a trade off between the degree of accuracy and the amount of computing time.

UGTRACE works on continuous curves or surfaces only. Tracing is stopped if the program encounters: 1) division by zero, or 2) values larger than the maximum value representable by the computer. Present implementation produces a partial scene description which UGTRACE has built up to the halting point. A separate logging algorithm is also available for debugging purposes.

1.3 Project profile

The UNIGRAFIX system is given "as is" without much programming documentation. It had to be installed in the McMaster environment [5], and required some minor tuning, before it was fully functional. Among the various supported output devices, those which are available at McMaster had to be configured and tested. These include the AED 512 colour graphics terminal, the IMAGEN laser printer and the IBM PC, which emulates a Tektronix

[5] BSD 4.3 UNIX running on the Computer Science and Systems VAX. Abbreviated CSSVAX hereafter.

4010 graphics terminal [6]. A general understanding of device independent graphics and the working principles of each device was necessary.

The rest of the project focused on the development of UGTRACE, the generator. It was concerned with translating a mathematical specification of a curve or surface into a UNIGRAFIX scene. Instead of delving into aspects like display algorithms and device interfaces, the development effort was spent on designing an input grammar, parsing algorithms, design and manipulation of transitional data objects, and tracing strategies. The user interface is required to be simple and straight forward.

All the code is written in the C programming language. The final version was compiled and run under the BSD 4.3 UNIX operating system on the department's VAX 11/780 computer. As for developing and testing, a micro-computer was also used. Code was mostly developed on the PC running under DOS [7] using the Lattice C compiler. After uploading to the mainframe, source codes were compiled using the CC compiler. For this reason, system dependencies of programs have been kept to a minimum.

A general introduction to the project is presented in this chapter. The rest of the report will be geared towards the

[6] The emulation is supported by QK-kermit, a communication package used at Queen's University, Kingston.

[7] DOS is a short form for Disk Operating System

software development activities. The concept of multi-device support and modifications done on the UNIGRAFIX system are discussed in Chapter 2. In Chapter 3, the design and implementation of UGTRACE is described. A brief summary on the project and recommendations on further development is presented in Chapter 4.

Chapter 2

MULTI-DEVICES OUTPUT SUPPORT

2.1 Device independent plotting

All renderers of the UNIGRAFIX system support multi-device output. A scene can be sent to a wide variety of output devices, ranging from plotters to CRT terminals. Whenever a renderer is called, the type of output device is specified as one of the run-time arguments. But this information is not used until the very final stage, when the picture is plotted to the display medium [1]. Data representation and manipulation, such as 3D to 2D view volume projection, rotations and other transformations, and hidden features removal, are done in a device independent co-ordinate system. In the final plotting stage, the proper device driver is selected from a library of drivers. The picture is mapped from device independent co-ordinates to device co-ordinates, and plotted to the display medium.

Conceptually, there are two levels of plotting (See figure 2.1). Since plotting mechanisms and command protocols differ from device to device, direct plotting would be difficult not only to implement but also to debug, to modify and to expand. Under such

[1] E.g. plotter paper and CRT screen.

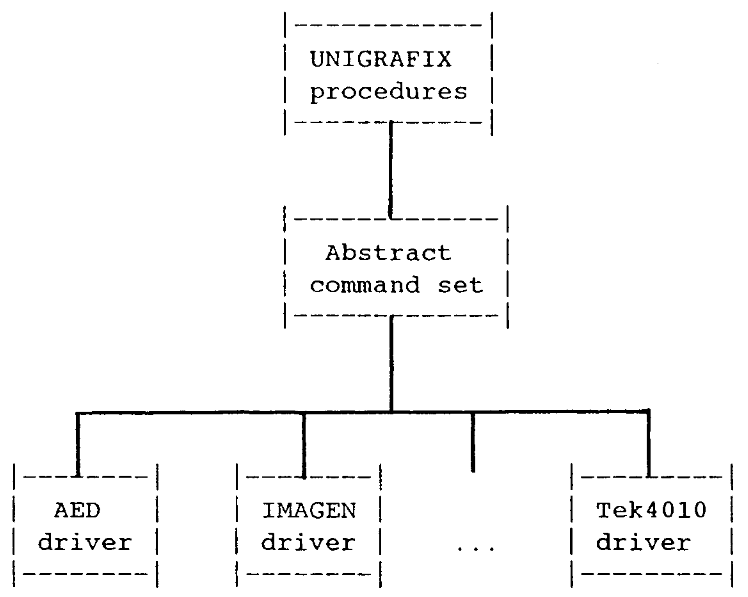


Figure 2.1 UNIGRAFIX output interface

considerations, another level is added on top of the device interface. In the higher level, plotting is done in a set of abstract functions. Each of these functions is a general command requesting a certain plotting action to be done. But the underlying mechanics of how the request is being achieved are transparent to the function. Specific information about the output device is not relevant on the abstract level. The interface level is basically a collection of device drivers. For every output device the renderer supports, there is a corresponding device driver. The drivers consist of concrete and device specific functions. These functions cause all the actions which are required to complete an abstract request. In this two level plotting scheme, new devices are supported readily as soon as the driver is available.

2.2 The abstract command set

The abstract command set contains five commands, three of which cause plotting actions. The other two are initializing and closing procedures. Similar to a file, a device has to be opened before any information can be sent, and closed after the plotting is completed. In between the opening and closing commands the viewport is drawn in a series of straight line segments. Curves are approximated by continuous short segments. Two of the plotting commands are defined to draw line segments. Character strings can also be placed in the picture for labeling purposes. However, the current version of UNIGRAFIX has no freedom on text orientation and character attributes yet.

Although the commands are capable of drawing only line segments, the viewport can be plotted in two different ways. In the first method, the viewport contents are broken down into a sequence of vectors. Each vector is defined by a starting co-ordinate, an ending co-ordinate and a set of line attributes such as colour and style. This method is suitable for describing points, edges, and boundaries. Wire frame pictures are plotted easily and efficiently under this method because they consist of only points and edges (See figure 2.2 (a)). In the second method, the viewport is stripped into a set of horizontal lines. Each line is partitioned in segments of different attributes. The entire viewport is scanned out sequentially from top to bottom, and horizontally from left to right. Because this continuous scanning format, solid displays, with hidden feature removed, shades, and colours are best plotted under this method (See figure 2.2 (b)). Although the two methods described are principally different from each other, they can be used simultaneously on one plotting. A picture, for example, can be plotted in solid display, while the edges are being highlighted in a different colour (See figure 2.2 (c)).

2.3 The AED 512 colour graphics terminal

The AED [2] 512 colour terminal is a powerful, stand alone, graphics workstation with keyboard and joystick integrated into it. Display resolution is 512 by 512 pixels with a total of 256 colours available. It comes with 21 Kilobytes of RAM [3] and a built-in 6502A microprocessor. When connected to a host computer, it can be used either as a telecommunications terminal or as a graphics peripheral. In the Interpretor Mode, graphics commands are encoded according to the AED Terminal Command Protocol [4]. Each alphanumeric character received at the

[2] AED is a short form for Advanced Electronic Design
[3] RAM is a short form for Random Access Memory
[4] Also known as AED Terminal Control Protocol. Ab-

terminal represents a request to perform some actions on the screen. If arguments such as co-ordinates and drawing colours are required, then they are given in binary numerals following the command code.

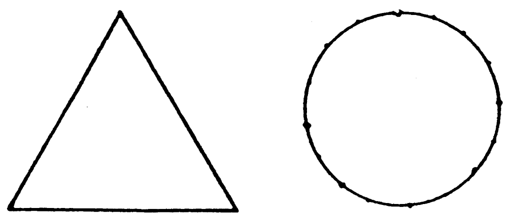
The AED terminal at McMaster is connected to the CSSVAX as a telecommunications terminal. Under UNIX, logged on terminals are assigned a logical device address. It is by this address the AED terminal is identified, and by this address TCP commands are sent. However, under the architecture of UNIX, a user may logon at several terminals at the same time. A user may want to run UNIGRAFIX on a control terminal and sent the output to another display terminal. It is, therefore, necessary to determine where the AED terminal is. UNIGRAFIX makes use of an environment variable called GRTERM, to store the address of the AED terminal. TCP commands are sent in raw mode [5] to the address at GRTERM instead of the address of the running terminal. However, it is the user's responsibility to set GRTERM to the correct device address. A small utility program called 'wtty' has been written to report all terminal addresses under the user's id.

The AED driver consist of five routines. Terminal communication is set to raw mode so that bytes received and sent are not inspected at the operating system level. During the

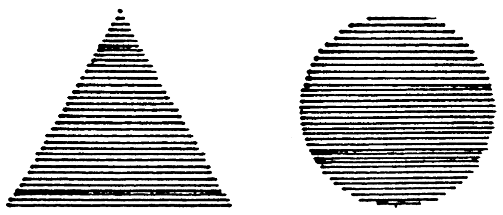
---

breviated TCP hereafter.

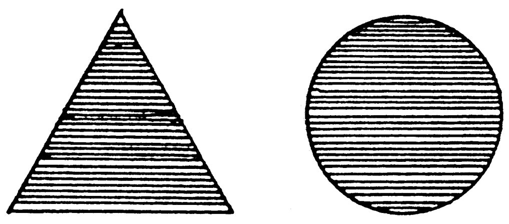
[5] In raw mode, bytes sent and received are not filtered and not interpreted by UNIX.



(a) Vector mode



(b) Raster mode



(c) Combined mode

Figure 2.2 UNIGRAFIX plotting modes

opening procedure, the AED terminal is reset to the Interpreter Mode. The colour map used by UNIGRAFIX is downloaded, and the raster size is initialized. Mapping and scaling constants are also initialized. Coordinates, if out of range, are clipped instead of wrapped around. Raster scanning commands and vector plotting commands may be interleaved. The exact situation depends on how the image is plotted and what underlying algorithm is used. Only the areas of interest are scanned out. Spaces in between are filled with the background colour. Two global variables are used to remember the scanner position. This position is used to determine the amount of background filling and to resume scanning after a vector plotting request is serviced. Texts are written by temporarily returning to Alphanumeric Mode. When imaging is finished, the closing procedure restores the AED terminal to a telecommunications terminal.

The original driver works well in most situations. Two terminal status inquiry routines, gtty() and stty(), were replaced by an equivalent updated version, ioctl(). It also happens that direct interleaving of raster scanning and vector plotting caused the image to distort, apparently due to byte loss during the interleaving. Solely raster scanning or vector plotting ran well without this problem. To correct the situation, vector plotting commands are now redirected to a temporary file instead of sending them directly to the AED terminal. Since the commands are alphanumerics, they are simply stored transitionally to an ASCII file. Text labelling is redirected as well. After raster scanning is completed, the file is piped to the terminal and is deleted afterwards. In this schema, raster scanning is given a higher priority with other interruptions suppressed.

2.4 The IMAGEN 3320 laser printer

The IMAGEN 3320 laser printer is a 20 page/min high speed, graphics printer. The command protocol used to drive the machine is called the ImPress language. Similar to AED's TCP, ImPress commands are also coded in 8-bit bytes. Each print job is an ImPress script made up of two parts. A job header is required to specify all kinds of parameters ranging from pen width to paper size. Fonts, styles and glyphs are selected from the printer's library. They can also be defined by the user. Following the header is the main body of the print. Besides running on ImPress, the printer also comes with several emulators. Command protocols of other types of printers, such as the Tektronix, are also allowed. However, UNIGRAFIX used ImPress.

Unlike a terminal, which belongs only to the logged on user, the printer is shared publicly with all users. Since the printer queue is maintained on the operating system level, UNIGRAFIX renderers do not drive the machine directly. All the plotting commands issued by UNIGRAFIX renderers ars written to a sequential file. When plotting is completed, UNIGRAFIX requests

UNIX to queue the script file for printing. The concrete set of commands for IMAGEN consists of opening and closing procedures, and one vector plotting command. Although IMAGEN has families of glyphs for shading and grey scales, the current version of UNIGRAFIX does not make use of this capacity.

The original version of the IMAGEN driver worked well except for minor errors in the job header section. The syntax of some initializing commands was not correct. These errors were corrected accordingly.

2.5 The TEK4010 terminal

The Tektronix graphics command protocol is popular for its simplicity. Many graphics peripherals, besides their own command sets, also emulate the Tektronix standard. Both AED and IMAGEN support Tektronix commands.

The Tektronix standard consist of two modes, alpha (text) and graphics. Under alpha mode, a Tektronix terminal receives and displays 8-bit bytes as ASCII characters. However, there are several non-displayable, control characters which switch the terminal in and out of alpha mode. GS, Group Separator, is the ASCII character 29. All the bytes received following GS are interpreted as point co-ordinates. The cursor is positioned to the first point. All successive points are linked together, resulting in a path on the screen. The path terminates upon receipt of another GS which means the opening of a new path, or upon receipt of another control character US, Unit Separator. The Tektronix protocol does not support colour variation; a pixel can only be on or off.

One bug was found in the original TEK4010 driver. When bringing the cursor to home position at the end of the plotting, a tail was always drawn from the last point to the home position. Upon investigation, it was discovered that an US control character was missing between the last point and the homing command. Thus the path was extended for an undesired section. This error was corrected by inserting the US character.

```
|------------------------------------------------------------------|
|                                                                  |
|                                                                  |
|   |---------------------------------|    |-----------------|   |
|   |                |---------------| |    |                 |   |
|   | |----------|   | |-----------| | |    |  |-----------|  |   |
|   | |  IBM PC  |<-->| Tektronix |<------------>| Tektronix |  |   |
|   | | terminal |   | | emulator  | | |    |  |  driver   |  |   |
|   | |----------|   | |-----------| | |    |  |-----------|  |   |
|   |                |               | |    |        |        |   |
|   |                |   QK-Kermit   | |    |  |-----------|  |   |
|   |                |---------------| |    |  | UNIGRAFIX |  |   |
|   |                                  |    |  |-----------|  |   |
|   |                                  |    |                 |   |
|   |               DOS                |    |      UNIX       |   |
|   |---------------------------------|    |-----------------|   |
|                                                                  |
|                                                                  |
|              Figure 2.3  PC/UNIGRAFIX interface                  |
|                                                                  |
|                                                                  |
|------------------------------------------------------------------|
```

In this project, IBM PCs are used to emulate Tek4010 terminals using QK-Kermit [6] (See figure 2.3). The software runs on the DOS operating system. Connection to UNIX is made via local area networks (LANs), or via modem by telephone dial-in. Although the PCs were able to receive Tektronix commands, resolution is being downgraded from Tektronix's 1024 by 780 to the PC's 200 by 640.

---

[6] A communication package developed by Mr. Victor Lee at Queen's University, Kingston.

Chapter 3

UGTRACE: A GENERATOR

3.1 Overview

The development of UGTRACE is made up of three sections. The input section consists of designing procedures to load and to parse instruction commands and trace specifications. The tracing section consists of designing procedures to sample parameter values and to calculate the corresponding point coordinates. And lastly, the output section consists of designing procedures to file the sampling of points in UNIGRAFIX plot description format.

From input to output, information is expanded and carried from one form to another. Highlights of internal data structures are binary tree representation of parametric equations and hierarchical linked list representation of curve and surface loci. Others include composite constructs from basic elements such as pointers and arrays. In the output section, temporary files are used to help reduce heap memory.

The program runs in batch mode. All information needed must be fed at the initial step. Specification text can either be read from a file or typed at the terminal. However, UGTRACE does not prompt the user for missing information. In case of errors, it reports the status and the corrective action. There is also an optional logging scheme implemented to facilitate the debug process.

3.2 Input section

UGTRACE has two levels of input, command line input and trace specification input. Trace specification is processed in two steps: a loading step and a parsing step. Beyond the input section, no user interaction is expected.

Since there are different modes in which UGTRACE runs, it is necessary that the user be able to select the mode he desires. In order to provide such freedom, UGTRACE accepts UNIX-like arguments given at the command line level. These arguments are interpreted as switches to turn running modes on and off. For example, the following command:

```
maccs 4 > ugtrace -fi sin_curve -fo sin_graph -a
```

specifies the input file to be sin_curve and the output file to be sin_graph. A frame of reference is also requested. The order of these switches are not important. Each switch has a default position. If a switch is not mentioned in the command line, the default is applied. Hence the user includes only the non-default switches of those modes he desired. Figure 3.1 is a summary of all available switches and their default positions.

```
-fi <filename>  pathname of specification text.
                Default input mode uses STDIN.

-fo <filename>  pathname of output file.  Default
                output mode uses STDOUT.

-us  unit step tracing.  Default tracing mode.

-ul  unit length tracing.

-dp  display patch (surface only).  Default display
     mode for surfaces.

-dm  display mesh (surfaces or curves).  Default
     display mode for curves.

-a   show axis.  Default axis mode shows no axis.

-l   logging.  Default log mode is no logging.

If switch is not included in command line, default
is applied.
```

Figure 3.1 Command line switches

The trace specification is a text file prepared using any text editor. The text includes the set of parametric equations to be traced and limits of the trace. Special keywords are placed at the beginning of each line to direct UGTRACE how to treat the line of text received. The following fragment gives a specification of one complete cycle of the sine function. Refer to figure 3.2 for the complete input syntax.

```
{ This is an example illustrating UGTRACE input }

constant  PI = 3.141592654
constant  TWOPI = 2 * PI

parameter  s, t
variable   x, y, z

x = t
y = sin (2 * PI + t)       { This is a sine curve }

domain    0 <= t <= PI
domain   PI <= t <= TWOPI
```

Comments are enclosed by "{" and "}". UGTRACE skips whatever text following "{", until it reads a matching "}". Comment text may embedded anywhere in the text or span several lines. Nesting of comments is accepted as long as the number of parenthesis match. Nested comments are kept track of using a counter. Unmatched opening or closing comment parenthesis results in a non-zero counter value.

The first category of information is constant macros. Special values or heavily used numbers are given "names", called identifiers. These identifiers are then used in the parametric equations. UGTRACE looks up the values of these identifiers, and uses them for evaluation. With this notion, lengthy constant expressions can be shortened in appearance, special values can be assigned meaningful names, and heavily used numbers can be modified or replaced easily. When used in an expression, names like "PI" are certainly more meaningful than 3.141592654. The CONSTANT statement declares and defines a constant identifier.

The following syntax is recognized by UGTRACE. Keywords are case insensitive and must be placed in the beginning of a line. Anything enclosed by [] is optional.

```
CONSTANT {identifier} = {token | expression}

PARAMETER {identifier} [ [, {identifier}] ... ]

DOMAIN {token} {lim} {parameter identifier} {lim} {token}
   [, [{token}] [, [{token}] ] ]

VARIABLE {identifier} [ [, {identifier}] ... ]

{variable identifier} = {expression}
```

where

{identifier} is the placeholder of an alphanumeric character string starting with a character,

{token} is the placeholder of a numeric value or a constant identifier,

{expression} is the placeholder of an unambiguous in-order expression (see figure 3.3 for detail), and

{lim} is the placeholder for one of one of the inequality signs "<" , "<=" , ">" or ">="

Figure 3.2 Syntax of input text

The declaration part causes an identifier to be reserved for assigning only constant values. The definition part assigns the identifier a numeric value. There are three ways of doing this. It can explicitly be a numeric value, an integer or a real number, as shown in the first constant statement in the input fragment. Or it can be another constant identifier whose value is already defined. Thirdly, it can be any mathematical expression whose format is described in figure 3.3. The expression may contain constant identifiers, as shown in the second constant statement in the input fragment. Other identifiers, such as parameter identifiers and variable identifiers which will be introduced shortly, are not allowed. Each CONSTANT statement initializes only one constant identifier. However, as many as sixty four CONSTANT statments are allowed in the current version.

The second category is parameter information. Each parameter is denoted by an identifier, and is given a domain of values. Tracing is in essence incrementing the parameter value within the given domain to obtain the locus of points. Both the PARAMETER statement and the DOMAIN statement are used to specify parameter information. The PARAMETER statement declares an identifier to be a parameter. The identifier is used later in the parametric equations. However, not all the declared parameters have to be used. In the input fragment, two parameters are declared but only one is used. For every used parameter, there must be at least one domain specified. With this set up, parameters used in the equations can be changed easily without significant modification of the input file. One PARAMETER statement may declare several parameters, or alternatively, several PARAMETER statements, each declaring one parameter, can be used. Up to eight parameters may be declared; however, only two of them may be used since a surface cannot have more than 2 parameters.

Domains are defined by the DOMAIN statement. Each statement defines only one domain which must be an interval. Besides the lower bound and the upper bound values, there are two other optional values in each statement. These values are used in controlling the tracing process. If they are not specified, default values deduced from the lower and upper bounds will be used.

The first optional value is a magnitude value. It is interpreted as the parameter step size under unit-step tracing, and as the required inter-point distance under unit-length tracing. Default magnitude for unit-step tracing is set to one percent of the current domain width. For unit-length tracing, this value is the distance associated with the first one percent of the domain. The second optional value is a tolerance value used only in unit-length tracing. For every trial step size, the distance between the projected point and the current point is calculated. If the difference between this distance and the required distance is smaller than this tolerance value, then the projected point is selected. Otherwise, a new trial value is tested. Tolerance is being set to one percent of trace magnitude if no override is given.

The current version of UGTRACE does not support discontinuity of curves and surfaces. In order to allow skipping over discontinuous areas, one parameter is allowed to have more than one domain associated with it. For example, if the curve or surface is discontinuous at t = 1, then this point can be skipped by defining two domain intervals,

```
domain  0 <= t <= 0.999
domain  1.001 <= t <= 2
```

This method of partitioning intervals is also useful for changing of tracing pace. For example,

```
domain  0 <= t <= PI, SIXTH_PI
domain  PI <= t <= TWO_PI, THIRD_PI
```

is equivalent to

```
domain  0 <= t <= TWO_PI, SIXTH_PI
```

except that the tracing pace is changed from SIXTH_PI to THIRD_PI. See Appendix C for an example of different tracing pace for same curve/surface.

Another way of skipping discontinuous areas is to specify open intervals instead of close intervals. Using the same example from the preceding paragraph, the domains can be specified

```
domain 0 <= t < 1
domain 1 < t <= 2
```

In tracing open intervals, UGTRACE shifts the actual tracing limits by 0.1 percent of the domain width. Thus in the example, the effect will be the same as using the previous method.

For each parameter, UGTRACE accepts up to thirty two DOMAIN statements. There is no checking against overlapping of intervals. Domains are traced in the order they are defined with no regard to their limits. Overlap areas are traced more than once.

The final category is variable information. Similar to the previous two categories, variables require declaration and definition. The declaration statement begins with the keyword VARIABLE. Several variables may be declared within one VARIABLE statement. Or equally good, several VARIABLE statements, each declaring one variable, may be used. Up to eight variables are accepted by UGTRACE, but only three can be defined. There is no keyword used for the definition of a variable; the parametric equation is used. However, the variable identifier must reside on the left hand side of the equation, and the expression on the right. Layout of the expression obeys exactly the same rules as an in-order expression. It can be a combination of numeric values, constants and parameters. The grammar of a valid expression is shown in figure 3.3. Operators are given the following precedence:

1) power "^",
2) trigonometric and exponential functions,
3) multiplication "*" and division "/",
4) addition "+" and subtraction "-",

with association from right to left. Parentheses are required.

```
<expr>       ::= <term> | <term><addop><expr>
<term>       ::= <fact> | <fact><multop><term>
<fact1>      ::= <func><fact2> | <fact2>
<fact2>      ::= <argu> | <argu><powop><fact2>
<argu>       ::= <lit> | <iden> | "("<expr>")"
<iden>       ::= <alphanumeric string>
<lit>        ::=  real  |  integer
<addop>      ::= "+" | "-"
<multop>     ::= "*" | "/"
<func>       ::= "sin"   | "cos"   | "tan"   |
                 "asin"  | "acos"  | "atan"  |
                 "sinh"  | "cosh"  | "tanh"  |
                 "asinh" | "acosh" | "atanh" |
                 "exp"   | "log"   | "log10" |
                 "sqrt"

<powop>      ::= "^"
```

Figure 3.3 Grammar used for expression

The order of the lines is not important. For example, a PARAMETER statement may precede a CONSTANT statement. However, an identifier must be declared before it is used. Keywords are not case sensitive. They also are not required to be typed out in full. The initial three letters are enough for PARAMETER, DOMAIN, and VARIABLE. For CONSTANT the initial five letters are required.

The complete specification text is read into memory buffers before any parsing is done. System I/O errors such as file nonexistence or unauthorized access are trapped. UGTRACE halts without wasting further computation efforts. Comments embedded in specification text are not relevant to the parsing step. They are filtered out in a pre-parsing process in order to reduce complexity of specification parsing.

Parsing is performed on a line by line basis. Each line of text is broken down into an array of words and mathematical symbols headed by a keyword. The keyword is used to denote the kind of information a line carries. For example, a line headed with the word "CONSTANT" means that the words in the current line are used to declare and define a constant identifier. Four other keywords are used for a complete plot specification. Input grammar is discussed in the next section. When a line is being analyzed, not only the syntax is parsed but also the content is validated against previous lines. The specification built is always a logical and consistent one. Mathematical expressions are represented by binary trees. Operators and functions are stored in parent nodes while operands and arguments are stored in children nodes. Before leaving this step, a general check is made to see whether there is information missing. Information specified but not used will also be dropped so that minimal information is passed on to the plotting step. It is designed in this way so that the input can be easily modified without a lot of retyping.

Figure 3.4 shows the domain lists and the expression trees built from the example fragment discussed earlier. Notice that unused information such as parameter t and variable z are removed

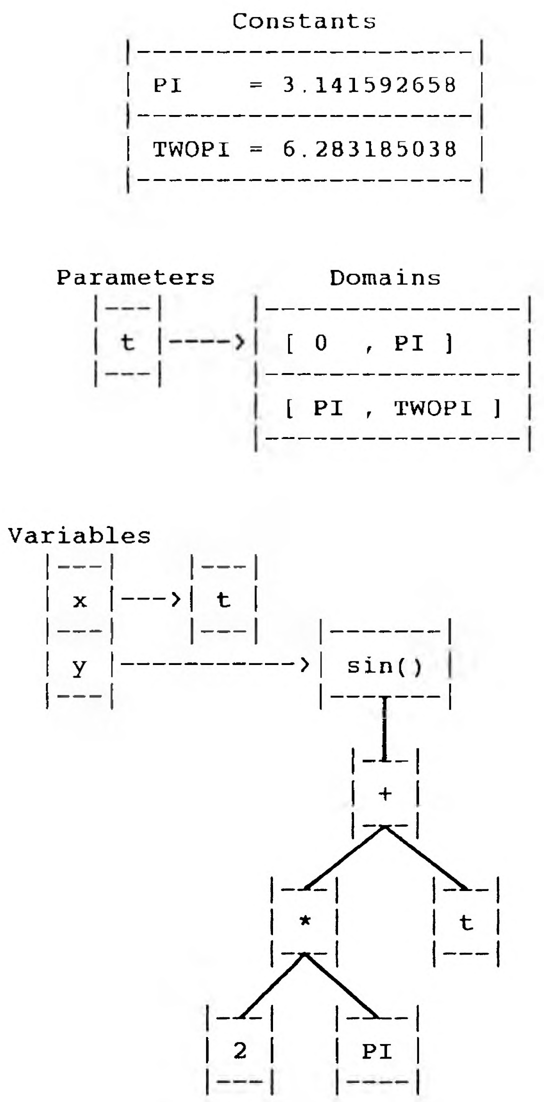


Figure 3.4 An example of UGTRACE internal representation, tracing section

upon exit from the parsing level. Macro constants are referred to their values via use of pointers. Records created in this section are heavily used in the tracing section.

3.3 Tracing section

Tracing is in essence the looping through given parametric domains, the evaluation of expression trees, and the recording of captured points. There are two different modes of looping. They are referred to as unit-step tracing and unit-length tracing from this point on.

In the first mode, unit-step tracing, the parameter is changed at uniform step size from one end of the domain to the other, until all the domains are traced.

```
for parameter_1 = lowerbound_1 , upperbound_1 , stepsize_1
                         .
                         .
                         .
```

In case of a surface where two parameters are involved, each parameter has its own step size. UGTRACE treats the surface as a family of curves. For example, if parameter u and v are used, then UGTRACE considers the surface as a family of curves whose parameter is v. Each member in the family has a unique value of u. Hence tracing becomes twofold: 1) picking of a member in the family, and 2) unit-step tracing of the selected curve. The process resembles two nested loops as follows:

```
for parameter_1 = lowerbound_1 , upperbound_1 , stepsize_1
  for parameter_2 = lowerbound_2 , upperbound_2 , stepsize_2
                              .
                              .
                              .
```

The size of increment is by default one percent of local domain width [1]. However, the user can override this default by specifying another value in an optional field in the DOMAIN statement.

Unit-length tracing is more complicated. It is also more time consuming. The step size, instead of remaining constant, is adjusted at every step so that the edges between successive points are of uniform length. Given the first point, the second point is located by trial and error. If the step size from the previous pair of points is available, it is used as the initial trial value. If no such value is available, then an arbitrary trial value is used. A point is located and the distance is computed. For sufficiently small trial step size, it is reasonable to assume that the distance to the trial point increases as the trial step size increases. Thus the trial step can be adjusted accordingly [2] by comparing the computed distance and the required length. Within several trials, the point is expected to home in to the correct position. However, the assumption is not guaranteed to be

[1] The width of a domain is defined as the difference between the upper bound and the lower bound.

[2] Trial steps are adjusted using the binary subdivision or expansion method.

true for all situations. It is possible for the step size to oscillate without converging. In some cases, it may even diverge. To provide an escape from these situations, UGTRACE imposed a limit on the number of trials for one step. Once the maximum number is reached, UGTRACE automatically stops tracing the current domain but continues with the next one, if there is any. In this way, a partial picture will be generated even if a total one is not possible. The required edge length is automatically set to one percent of the domain width. The user can override this default length by specifying another value in an optional field in the DOMAIN statement.

Unit-length tracing of surfaces is slightly different from unit-length tracing of curves. For curves, there is one parameter whose step size has to be adjusted in order to locate a point. The algorithm can be summarized as follows:

```
Set parameter_1 = lowerbound_1
Compute current point
Set initial trial step
While (parameter_1 < upperbound_1) do
    Repeat
        Compute trial point
        If (trial point distance != required edge length) then
            Adjust trial step size
            by binary subdivision or expansion
        If (no. of trials > max. no. of trials allowed) then
            Abort current domain of parameter_1
    Until (trial point distance = required edge length)
    Increase parameter_1 by trial step size
```

As for surfaces, two parameters are used. Similar to unit-step tracing, UGTRACE also sees the surfaces as families of curves, each member having an unique value of the first parameter. The criteria used to select the members to be traced is that the distances between the first points of successive members must be constant (Figure 3.5). The algorithm can be summarized as follows:

```
Set parameter_1 = lowerbound_1
Unit-length trace parameter_2, keeping parameter_1 constant
Set initial trial step size for parameter_1
While (parameter_1 < upperbound_1) do
    Set parameter_2 = lowerbound_2
    Repeat
        Compute trial point
        If (trial point distance != required edge length) then
            Adjust trial step size
            by binary subdivision or expansion
        If (no. of trials > max. no. of trials allowed) then
            Abort current domain of parameter_1
    Until (trial point distance = required edge length)
    Unit-length trace parameter_2, keeping parameter_1 constant
    Increase parameter_1 by trial step size
```

3.4 Output section

The tracing procedures do not produce the points directly in UNIGRAFIX format, but rather, a hierarchy of records is built (See figure 3.6). This structure is then converted to UNIGRAFIX format in output section.

The hierarchy results from surface tracing consists of three levels. At the highest level is a linked list of surface records (S1, S2, ... in figure 3.6). Each surface record represents a patch which is determined by one combination of domains of the two parameters. Associated to each surface node is

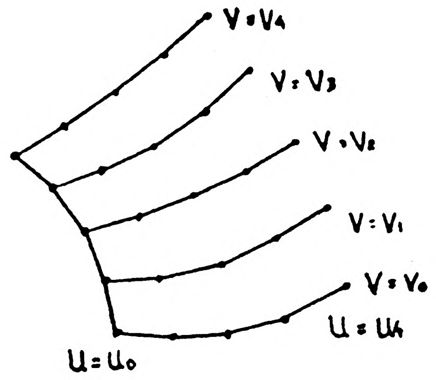


Figure 3.5 Scenario for unit length tracing of parametric surfaces

```
|----|    |-----|    |------|    |------|    |------|
| S1 |--->| C11 |--->| P111 |--->| P112 |--->| P113 |--> ...
|----|    |-----|    |------|    |------|    |------|
  |          |
  |          |
  |       |-----|    |------|    |------|    |------|
  |       | C12 |--->| P121 |--->| P122 |--->| P123 |--> ...
  |       |-----|    |------|    |------|    |------|
  |          |
  |          .
  |          .
  |          .
  |
|----|    |-----|    |------|    |------|    |------|
| S2 |--->| C21 |--->| P211 |--->| P212 |--->| P213 |--> ...
|----|    |-----|    |------|    |------|    |------|
  |          |
  .          .
  .          .
  .          .
```

Figure 3.6 UGTRACE internal representation, output section

a linked list of wire records (C11, C12, ... in figure 3.6). Each wire record represents a curve in the second parameter, with the first parameter constant. At the lowest level, each curve is represented by a linked list of point records (P111, P112, P113, ... in figure 3.6). Each point record contains the geometric coordinates and their corresponding parameter values.

The hierarchy for curves has only two levels. Each domain interval is represented by a wire record, with which a list of point records associated. If more than one interval is traced, then the curve is represented by a list of wire records.

Records of all three levels have a special tag for error indication. In case of a partial plot where some domains are not completed, tags are flagged from the point level up to the surface level.

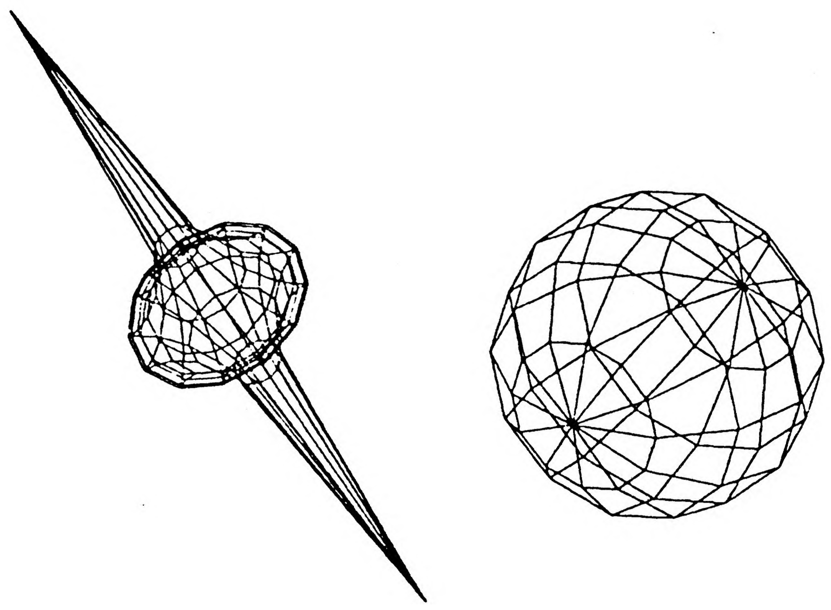
Three UNIGRAFIX entities, vertices, wires and faces, are used. A vertex is the coordinate description of a point in three space. Wires and faces are both lists of vertices, but the vertices of a face have to be coplanar. Since vertices are referenced by wires and faces, each of them is given a unique label. UGTRACE assigns labels in hexadecimal numbers. Detailed discussion of vertex, wire and face will not be pursued here. A general syntax is shown in figure 3.7.

Conversion is made patch by patch, one at a time. Two display modes, mesh mode (-dm) and patch mode (-dp), are

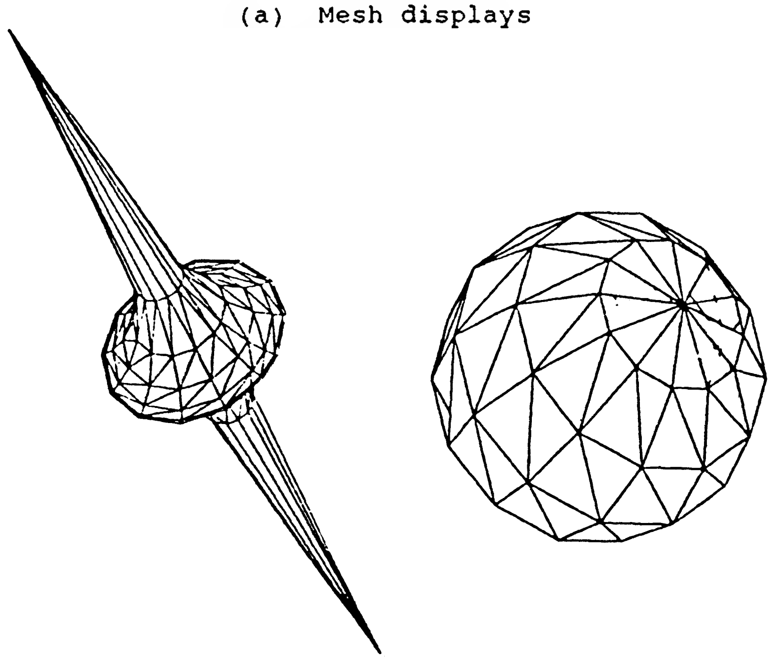
```
vertices:     v  ID  x  y  z ;

wires:        w  [ID] (v1 v2 ... vn) (...) [colorID] ;

faces:        f  [ID] (v1 v2 ... vn) (...) [colorID] ;

color:        c  colorID  intensity  [hue [saturation]] ;

comments:     {  [ anything {nesting is OK}
                   but unmatched [ or } ]   }
```

Figure 3.7 UNIGRAFIX syntax, abridged summary

available. Under mesh mode, the surface is converted to a framework of wires. There are no hidden elements. All vertices are shown. This is not the same when the surface is converted under patch mode. Stripes of surfaces between successive contours are cut into triangular patches. Since vertices of all triangular patches are coplanar. Surfaces of any curvature can be converted to a collection of faces. The picture under patch mode is a solid one. Faces blocked from view point by other faces will not be shown. Lighting and shading provides even more depth to the picture. Figure 3.8 shows a surface displayed under the two modes, and hence contrasts the difference. Patch mode is not available for displaying curves.



(a) Mesh displays



(b) Patch displays

Figure 3.8 UGTRACE display modes

Chapter 4

SUMMARY

The work of this project was divided into two parts: 1) testing and debugging of UGDISP, the latest renderer of UNIGRAFIX, and 2) designing and coding of UGTRACE. Combined together, these two parts form a displaying tool for parametric curves and surfaces.

The Berkeley UNIGRAFIX renderer, UGDISP, consists of over 12,000 lines of C code in 44 files. The size of the executable file is about 234 kilobytes. The various drivers are approximately 2,000 lines of C code. For part one, the UNIGRAFIX language and the UGDISP input/output interface were studied; the AED, the IMAGEN and the TEK4010 device drivers were tested for performance. Bugs discovered were corrected. It was also found that although the UNIGRAFIX language supports colour and light source specification, some device drivers were not completely developed to handle these entities. For example, if a surface is displayed on the IMAGEN printer, the faces are always left blank disregarding colour and light source. Other than this, UGDISP performs satisfactorily.

For part two, a piece of software UGTRACE was developed. UGTRACE is about 3,500 lines of C code. The size of the executable file is approximately 65 kilobytes. From an initial thought to the final product, the software engineering approach was followed closely. Objectives were defined in the analysis stage. After that, a blueprint of the solution was drafted. Then different sections were functionally outlined. The input section was built first, following by the tracing section and the output section. Testing was performed on an incremental basis. Whenever a section was finished, it was integrated into the existing system for testing. Enhancement and modifications were constantly made as new insights were discovered along the building and testing process. As a result, the blueprint changed from time to time. However, the objectives were not changed.

When both parts were completed, a set of trial runs were used to verify program correctness. Curves and surfaces with known shapes and forms were fed to UGTRACE. Output was piped to UGDISP for display. If debugging was necessary, a logfile was available through log mode for tracing of all intermediate values.

The rendering of curves and surfaces with UGTRACE and UGDISP is an iterative process. The user plays an important role in selecting domains, iteration step size, view points, view directions, and so on. By carefully choosing several views of the same curve or surface, the user can get a better idea of its actual shape.

The scope of this project covers up to three dimensional curves and surfaces given in parametric forms. The current version of UGTRACE accepts one specification at a time and generates one scene file for every specification. For future enhancements, a tracing shell can be developed so that it is possible to generate scene files of multi-curves and surfaces. Another potential development is to allow input of curves and surfaces given in implicit forms f(x,y,z), such as

```
x^2 + y^2 + z^2 = r^2
```

rather than in parametric forms

```
x = r * sin(a) * cos(b)
y = r * cos(a) * cos(b)
z = r * sin(b)
```

The restriction of three dimensional space can also be extended to four dimensional space. In such case, projections of four dimensional forms will be traced and displayed.

BIBLIOGRAPHY

COMPUTER GRAPHICS

[FOLE84] Foley, J.D. and Van Dam, A., Fundamentals of Interactive Computer Graphics. Addison-Wesley, July 1984.

UNIGRAFIX

[SEQU83a] Sequin, C.H. and Strauss, P.S., UNIGRAFIX. Proc. 20th Design Automation Conf.(pp.374-381), Miami Beach, FL, June 1983.

[SEQU83b] Sequin, C.H., et al, UNIGRAFIX 2.0 User's Manual and Tutorial. Tech. Report (UCB/CSD 83/161), U.C. Berkeley, Dec. 1983.

[SEQU83c] Sequin, C.H., Creative Geometric Modeling with UNIGRAFIX. Tech. Report (UCB/CSD 83/162), U.C. Berkeley, Dec. 1983.

[SEQU85] Sequin, C.H., The Berkeley UNIGRAFIX Tools Version 2.5. Technical Report (UCB/CSD 86/281), U.C. Berkeley, Dec. 1985.

PROGRAMMING

[KERN78a] Kernighan, B.W. and Plauger, P.J., The Elements of Programming Style (2nd ed.). McGraw-Hill, 1978.

[KERN78b] Kernighan, B.W. and Ritchie, D.M., The C Programming Language. Prentice-Hall, 1978.

[JAME77] James, M.L., et al, Applied Numerical Methods for Digital Computation (2nd ed.). Harper & Row, 1977.

[MCGI83] McGilton, H. and Morgan, R., Introducing the UNIX System. McGraw-Hill, 1983.

[MYER79] Myers, G.L., The Art of Software Testing. Wiley, 1979.

[SAHN85] Sahni, S., Software Development in Pascal (1st ed.). The Camelot Publishing Co., 1985.

[TREM84] Tremblay, J.P. and Sorenson, P.G., An Introduction to

Data Structures with Applications (2nd ed.). McGraw-Hill, 1984.

MANUALS

[AED] AED 512 Color Graphics Terminal, Terminal Control Protocol.

[IMAG] ImageServer XP Programmer's Guide, Part I and II.

[LATT85] Lattice C Compiler for 8086/8088 Series Microprocessor, Document Revision 2.15A. Lattice Inc., 1985.

[TEK83] Tektronix 4105 Programmer's Guide. Sep. 1983.

THESIS WRITING

[TURA67] Turabian, K.L., A Manual for Writers of Term Papers, Theses, and Dissertations (3rd ed. revised). University of Chicago Press, 1967.

[BROO75] Brooks, F.P.Jr., The Mythical Man-Month: Essays on Software Engineering. Addison-Wesley, 1975.

Appendix A

USER MANUAL

A.1 ugdisp

NAME

ugdisp - render a UNIGRAFIX scene on a screen or plotter

SYNOPSIS

```
ugdisp [ options ] [< scene]
```

DESCRIPTION

Ugdisp can render a scene on many possible output devices. Scene description is read from standard input, unless the -fi option was specified. The display is controlled by the following groups of options:

Viewing Geometry

-ep x y z
: Eye point for perspective view from this point.

-ed x y z
: Eye direction for parallel projection.

-vc x y z
: View center for a perspective view; i.e., the display is centered at that point in the scene.

-va angle
: View angle for a perspective view; must be between 0 and 180, exclusively. It defines the maximum angle of a square-based viewing pyramid, anchored at the eye point. The default view angle is 90 degrees.

-vr angle
: View rotation. By default the y-axis points up; the displayed scene is rotated CCW around the viewing direction by angle degrees.

Ugdisp centers the scene and scales it to the maximal size that would still fit in the rectangle of the screen or plot. Specifying view center or view angle for a perspective view overrides this auto scaling, and the picture may occupy only part of the screen or plot. If no eye direction or eye point is specified, the default view is -ed 0 0 -1, i.e., an orthogonal projection from the negative z-axis. Clipping is not performed with hidden feature removal, so the user should be careful when specifying a perspective view not to position the eye point too close (or inside) the scene. This will hopefully be remedied in the future.

Display Modes

-hn Hide nothing, make no visibility checks (Default).

-hb Hide back-faces, i.e., faces with face normal pointing away from eye.

-ho Hide overlaps; remove all features hidden by overlaps. Implies -hb.

-ab Add back-faces. Overrides any specific or implied -hb.

-se Show edges and wires only (Default).

-sf Show only faces without edges or wires. Implies -ho and -hb.

-sa Show faces and edges. Implies -ho and -hb.

-fw x y z d1 d2
Fade against white background in the interval d1 - d2 .

-fb x y z d1 d2
Fade against black background in the interval d1 - d2 . x, y and z specify the eye point; d1 and d2 are distances from this eye point.

-sg Show smoothly shaded faces (with Gouraud shading). Implies -sf.

If the -sa option is combined with gouraud shading then only a subset of the edges is displayed; those edges are wires, contour edges (edges with faces on one side only), intersection edges, and edges with one face on each side and with a dihedral angle that is less then the value of some specified corner angle. This corner angle defaults to 100 degrees, and can be changed with the following option:

-ca angle
The corner angle is set to angle degrees. The default is 100 degrees.

-st Show text. The first text statement in the input scene is executed.

-sc Show coordinate axes.

-in Detect and correctly display intersecting faces.

-w Perform extra checks to display warped (non-planar) faces.

Labeling

-lv Label vertices. The vertex identifier is printed next to the vertex position. If -hb was specified (or implied) then vertices that belong to back-faces are not labeled.

-lf Label faces. The face identifier is printed on the center of gravity of the face. Faces are not labeled if hidden features are removed.

-lw Label wires. The wire identifier is printed on the center of gravity of the wire.

-le Label edges with their length.

-la Label all (vertices, faces, wires, edges).

Identifiers starting with the character '#' are not printed. If an identifier contains a '#' then only the suffix following the last '#' is printed. Some devices do not support labeling.

Output Devices

-dv Output device is a Benson Varian plotter.

-dw Versatec 36'' wide-bodied plotter.

-dm Imagen printer.

-da AED 512 color display (set GRTERM to appropriate /dev/tty??).

-dx Vectrix color display (set GRTERM to appropriate /dev/tty??).

-dr IRIS graphics terminal (set GRTERM to appropriate /dev/tty??).

-di Ikonas frame buffer. A raster file called ``rast.iv'' is created, and can be displayed with the iv program.

Output is also sent to a variety of display terminals that usually serve as the user's console or tty. If the environment parameter TERM is set to the terminal's name then no device option is necessary. Otherwise (or when ugdisp is used from ugi with a permanent device option different from the console) the terminal type should be specified:

-dt Tektronix 4115 (TERM = 4115).

-dT Tektronix 4691 plotter.

-dk Tektronix 4010 (TERM = 4010).

-dK Tektronix 4107 (TERM = 4107).

-dh Hp 2648a (TERM = hp2648a).

-dS Sun microsystems workstation (Default, TERM = sun). If hidden feature removal is performed the picture is sent to the screen in 10 or more equal chunks. If only edges and wires are plotted then the picture is sent to the screen only when it is complete.

-ds Sun microsystems workstation. If hidden feature removal is performed the picture is sent to the screen scanline by scanline (it is slower than with the default -dS and useful only for debugging purposes). If only edges and wires are plotted then the picture is sent to the screen line by line; this is useful to preview complicated scenes before doing the hidden feature removal.

-df frame-file
Sun microsystems workstation. Frame-file is a name of a raster file that will contain the picture. The raster dimensions are 256 x 256 and it is meant to be an input for the framedemo program. The uganimate program allows easy creation of simple animations with framedemo.

If no device option is specified, and the TERM parameter is not set to any of the above terminals then a dumb terminal is assumed; the output is in a crude form of ascii characters.
The output to the plotters and to some of the terminals can further be controlled by setting its size:

-sx number
x-size of the plot is adjusted to fit into number inches. The default is the width of the display device.

-sy number
y-size of the plot is adjusted to fit into number inches. The default is the height of the display device. On the Varian and Versatec plotters the default is 8'' and 36'' respectively; specified y-size can be up to twice the default.

Files

-fi input-file
Read input scene from file input-file.

-fc command-file
Read options from file command-file.

-cm colormap-file
Read color map description from file colormap-file.

-fr raster-file
Put raster file in file raster-file. Raster files are named ``/usr/tmp/ug??????'' by default. This is useful if there is not enough space in /usr/tmp on your machine.

-kf Keep raster file. By default raster files are deleted after plotting; with this option the raster file name is printed on standard error and the file is not deleted. This is useful if you want to plot several copies of the same scene.

-kt Keep the temporary files that are created during the processing of a text statement. Those files are ``/usr/tmp/ug?????.tex'', ``ug?????.log'' and ``ug?????.dvi''. The default is to delete them.

-np No plot. The raster file is not sent to the plotter.

File-names can be expressed with all the csh conventions except globbing, i.e., start with ``~user/...'' or ``~/...'', contain environment parameters like ``$WORKDIR'', contain ``$$'' etc. If the file is not found in the current directory, and the -fi or -cm options are used, then ugdisp tries to read ``~ug/lib/input-file''.

The -fr , -kf and -np options apply only to the Varian and Versatec plots.

EXAMPLE

```
ugdisp -ep -1 2 -10  -va 30  -sa -in -dw -sy 8 < scenefile
```

FILES

```
~ug/bin/ugdisp
~ug/src/ug2/ugdisp
/usr/tmp/ug??????
```

SEE ALSO

ugi(UG), ugisect(UG), ugshow(UG), ugplot(UG), uganimate(UG)

DIAGNOSTICS

Ugdisp prints to standard error the elapsed user and system times (in seconds) after each step in the processing. This printout is suppressed on some devices where is would interfere with the picture, and when ugdisp is called from ugi. If the -v option is specified, ugdisp will print more detailed statistics: number of intersections in -in mode, number of warped faces in -w mode, and the cpu times of the hidden feature removal module.

BUGS

Does not clip to the viewing pyramid in perspective view, so behavior is unpredicted if the eye-point is too close or inside the scene.
Horizontal edges may be excessive or missing in -ho and -sa modes.

AUTHOR

Nachshon Gal

A.2 ugtrace

NAME
    ugtrace - trace a parametric curve or surface

SYNOPSIS
    ugtrace [options] [< source file] [> output file]

DESCRIPTION
    Ugtrace can parse a parametric specification of a curve or surface, and translate it into the UNIGRAFIX scene descriptive language. Specification is read from standard input, unless -fi option was used. The following run-time options are implemented.

Input and Output
    -fi source filename
        Read specification from source file. This switch should not be used in combination with input redirection. Default is stdin.

    -fo output filename
        Write scene description to output file. This switch should not be used when output redirection is used. Default is stdout.

Tracing Mode
    -us unit step tracing. Parameter is changed at uniform step size. Default tracing mode.

    -ul unit length tracing. Also known as unit speed tracing. Successive points plotted will be at equal distance apart.

Display Mode
    -dp display patch. Surface only. Solid view of surface is shown. Hidden features are removed under this mode. Default for surface display.

    -dm display mesh. Function is displayed as a wire mesh.

Miscellaneous Options
    -a show axis. Include coordinate axes into scene description. Default is no axis.

    -l logging. Mainly for debugging during development.

Values of internal data variables at selected checkpoints are logging to a temporary file. Default is no logging.

EXAMPLE

```
ugtrace -a -fo spec.out < specification
```

SEE ALSO

ugdisp(UG)

DIAGNOSTICS

Execution is divided into five blocks. UGTRACE reports on the control terminal the execution status. Statistical data on each block is also printed. If UGTRACE is not able to complete a trace, then the reason for failure will be reported.

BUGS

A surface patch generated by UGTRACE may not be always visible even if nothing is blocking the view. This is because UNIGRAFIX considers one side of a face as inside and the other side as outside. Only the outside is visible. Therefore, if the viewpoint is looking at the inside of a face. It will not be displayed.

AUTHOR

Henri Cheung

Appendix B

TUTORIAL

Introduction

The purpose of this tutorial is to teach the user how to display parametric curves and surfaces using the UNIGRAFIX system. We will go through an example of a spiral about the z axis,

```
x = u * sin(u)
y = u * cos(u)
z = 10 * u

for 0 <= u <= (10 * pi)
```

in doing so. We will learn to use the generator UGTRACE and the renderer UGDISP. The user is assumed to have a general knowledge of the UNIX operating system, and the authority to access the UNIGRAFIX files. A graphics terminal is also required if the user intends to display the scene to the screen.

The tutorial is divided into three parts:

1) creating the specification file,
2) generating the scene file using UGTRACE, and
3) viewing the scene file using UGDISP.

Creating the Specification File

The specification file has to be prepared manually by the user. Any text editor can be used. The syntax and semantics of the specification is simple and straight forward. Five types of statements are used. We shall discuss them in the order that they appeared in the specification. Individual statements must be completed within one line.

The PARAMETER statments

In our example, the parameter is named u. The PARAMETER statement is used to declare one or several parameter names. In other words, we are reserving the names as parameter identifiers. Any alphanumeric string starting with an alpha character can be used as a parameter name. Unlike the leading keyword, names are case sensitive. The name u is not the same as the name U. To declare u, we enter the following line:

```
        parameter u
```

If more than one identifier is declared within one statement, then the identifiers are separated by a comma. The leading keyword "parameter" can be typed either in lowercase or uppercase (like all other keywords discussed later). A short form "par" is also acceptable. Although only two parameters are used in surface specifications, each specification file accepts up to 8 parameter declarations.

```
        PARAMETER p1,p2,p3,p4,p5,p6,p7,p8

        parameter p1,p2,p3,p4,p5,p6,p7,p8

        parameter p1, p2, p3, p4
        PAR p5, p6, p7, p8

        par p1
        par p2
        par p3
        par p4
        par p5
        par p6
        par p7
        par p8
```

All of the above four groups of PARAMETER statements are correct and semantically identical. There are no restrictions on how many times the PARAMETER statment can be used in one specification. However, a parameter must be declared before it can be used in the equations and the DOMAIN statments discussed later.

The VARIABLE statments

Next, we have to declare the variables in our example. They denote the Cartesian coordinates of the points being plotted in 2-space or 3-space. Declaration of variable names is identical to that of the parameters, except that the leading keyword is changed to "variable" (or "var" in short). In our example, we enter

```
        variable x, y, z
```
or
```
        var x, y, z
```

Although only two variables are used in curves specifications, and three in surfaces, each specification file accepts up to 8 variable declarations. The user must be careful not to use names that are already declared.

The parametric equations

After all the parameters and variables are declared, we can define the parametric equations for the curves or surfaces which we want to display. No special keywords or particular grammars are required. The equation itself is entered into the specification file. However, parentheses should be used to avoid ambiguities. Each specification allows up to 3 parametric equations. For our example, we copy directly from the equations above,

```
x = u * sin(u)
y = u * cos(u)
z = 10 * u
```

Note that a declared variable must be in the leading position followed by an equal sign "=". The expression on the right consists of operands and operators. Operands can be integers, real numbers or declared parameters. The operators are listed in decreasing precedence as follows:

power operator	^
pre-defined functions	sin(), cos(), tan(), asin(), acos(), atan(), sinh(), cosh(), tanh(), asinh(), acosh(), atanh(), exp(), log(), log10() and sqrt()
multiplication and division	* , /
addition and subtraction	+ , -

The DOMAIN statments

Now that we have declared and defined the variables and the equations, we specify the domains of the parameters. Each DOMAIN statement starts with the keyword "domain" (or "dom"), followed by the bounds of the domain. In our example, we want to display parameter change for u from 0 to ten pi. The required DOMAIN statement would be:

```
dom 0 <= u <= 31.41592654
```

The bounds can be in decreasing order. The following statement is equally valid:

```
dom 31.41592654 >= u >= 0
```

For open intervals which do not include end points, inequality symbols "<" and ">" are used instead of "<=" and ">=" shown.

In the DOMAIN statements, there are two other optional fields whose values are used to control of the trace process. The first field is used as a magnitude value. In unit-step tracing, this value is interpreted as the step size for parametric increment. In unit-length tracing, the value is used as the required edge length for the scene to be generated. The second value is used in unit-length tracing only. It is the tolerance value for the required edge length. In most cases, the user need not worry about these values. The system will calculate a default based on the domain's upper and lower bounds. However, if we enter values into these optional fields, the default values will be overridden. For example,

```
        dom 0 <= u <= 31.41592654 , 0.25 , 0.005
```

In the above statement, we specify the magnitude of u to be 0.25 and the tolerance to 0.005. In unit-step tracing, the value of u will be increased from 0 at steps of 0.25 until ten pi, and 0.005 is ignored. In unit-length tracing, all the edges in the scene will be with the range (0.25 + 0.005) and (0.25 - 0.005).

The CONSTANT statments

The previous four types of statements are sufficient to define a specification file for UGTRACE. However, we notice that some numerical values such as pi and e have special mathematical meanings, and are used more often than others. It is more appropriate and more readable to use their names than to use their values. The CONSTANT statements are designed for such purposes. Each CONSTANT statement starts with a leading keyword "constant" (or "const" in short), followed by a constant name and the equivalent value. pi and e would be defined by the following statements,

```
        const pi = 3.141592654
        const e = 2.7182818
```

Note that both declaration and definition of a constant name is included in the same statement. Each statement can declare and define only one constant. The value on the right hand side of the equal sign can be expressed in an expression format similar to that of the parametric equations. However, parameter names are not allowed in the expressions. In our example, we may define pi and ten pi to be constants:

```
        const pi = 3.141592654
        const tenpi = 10 * pi
```

With the above constants defined, our specification file becomes

```
        x = u * sin(u)
        y = u * cos(u)
        z = 10 * u

        dom 0 <= u <= tenpi
```

The comments

Comments are any text enclosed by "{" and "}". They can appear anywhere within the specification. Nesting of comments is allowed, but the number of "}" must match the numbers of "{". With this feature, we can place explanatory notes in the specification file. Comments can also be used cross out information from the specification without deleting the text from the file. In practical application, the specification file of our example may look like this:

```
        {************************************************}
        { This is an example used for tutoring purpose.  }
        { File name of this file is called "scene.spec"  }

        const pi = 3.141592654         { constant used   }
        par u                          { parameter used  }
        var x, y, z                    { variables used  }

        x = u * sin(u)
        y = u * cos(u)
        {
        z = 10 * u                  { original version of z }
        }
        z = (pi - u)^2              { modified version of z }

        {************************************************}
        const tenpi = 10 * pi          { constant used   }
        const a     = 0.25             { constant used   }
        const b     = 0.005            { constant used   }
        {************************************************}
        dom 0 <= u <= tenpi{, a, b}    { uncomment       }
                                       { for override    }
        {************************************************}
```

Generating the Scene File

After the specification file has been created, we use

UGTRACE to generate the scene file. We will go through this section by giving some examples:

```
        ugtrace -fi scene.spec -fo scene.script
```

The above command specifies an input specification file called "scene.spec" and generates a scene file called "scene.script". Since no tracing mode is specified, UGTRACE uses unit-step automatically. Under this mode, points generated are at equal parameter step size. Since "scene.spec" contains a surface specification, default display mode is patch display.

```
        ugtrace -a -ul -dm < scene.spec > scene.script
```

The above command uses redirection instead of I/O arguments. It specifies the axes to be included into the scene file. Since -dm is used, display mode will be display mesh. Tracing mode is overridden to be unit-length tracing. Under unit-length tracing, all the edges generated in the scene file will be of equal length.

The usage of UGTRACE is like other UNIX system commands. The command name is followed by a list of arguments. Each argument starts with "-". For a summary of the arguments, the user can refer to the UGTRACE user manual for detail.

Viewing the Scene File

After the scene file has been generated, we are finally in a position to display the curves or surfaces we started with. We use UGDISP to view the scene. Three display devices are available at McMaster University. They are the AED terminal, the IMAGEN laser printer, and the IBM PC's which emulate a Tektronix 4010 terminal using QK-Kermit. In this section we will focus on how to send output to these devices. For a complete tutorial on how to use ugdisp, see UNIGRAFIX 2.0 User's Manual and Tutorial by C. H. Sequin, and also the UGDISP user manual.

Using the AED 512 terminal

First, we log on at the AED terminal. Then, we find out the system address of the AED terminal. Type, on the AED terminal, the command

```
        tty
```

and the system will display the address of the terminal, for example:

```
        /dev/ttyA4
```

Ugdisp make use of a system variable called GRTERM to identify the address of the AED terminal. Therefore, we have to set GRTERM before we run ugdisp. To set GRTERM, type the command

```
        setenv GRTERM=/dev/ttyA4
```

Once GRTERM is set, we can run ugdisp on either the AED terminal or another control terminal. Ugdisp will send the output to the address obtained from GRTERM. However, we must specify that the output device is an AED terminal using the "-da" switch. To display scene.script, type the following command,

```
        ugdisp -da < scene.script
```

and the output will be sent to the AED terminal.

Using the IBM PC's

In order to use the PC's as a display terminal, we must obtain a copy of QK-Kermit, a communication software package running under DOS. After booting the PC using a DOS system disk, we run QK-Kermit to dial in CSSVAX and log on. Terminal type is VT100. Once logged on, we run ugdisp by typing the following command

```
        ugdisp -dk < scene.script
```

and the output will be sent to the PC. The "-dk" switch specifies a Tektronix terminal. Under this setting, the control terminal and the display terminal must be the same terminal.

Using the IMAGEN printer

Any logged on terminal can be used to send the scene output to the IMAGEN printer. The "-dm" is used. To obtain a hardcopy of the scene, type the following command,

```
        ugdisp -dm < scene.script
```

and the output will be sent to the IMAGEN printer linked to the CSSVAX.

Appendix C

EXAMPLES

```
{
  Example C.1  Unit-step Versus Unit-length

  This example demonstrates the difference
  between unit-step tracing (-us) and unit-
  length tracing (-ul)
}

par u, v
var x, y, z

x = u
y = sin(u) * sin(u) * sin(u) * sin(u)
z = v

const pi  = 3.141592654
const pi2 = 2 * pi

{ Use this block for unit-step tracing }
{
const ustep = pi / 6
dom 0.0 <= u <= pi2, ustep
dom 0.0 <= v <= 6.0, 0.6
}
{ Use this block for unit-length tracing }

dom 0.0 <= u <= pi2, 0.4
dom 0.0 <= v <= 6.0, 0.6

{
  Comment:

  Note that the grid sizes are more regular
  on the plotting resulted from unit-length trace
}
```

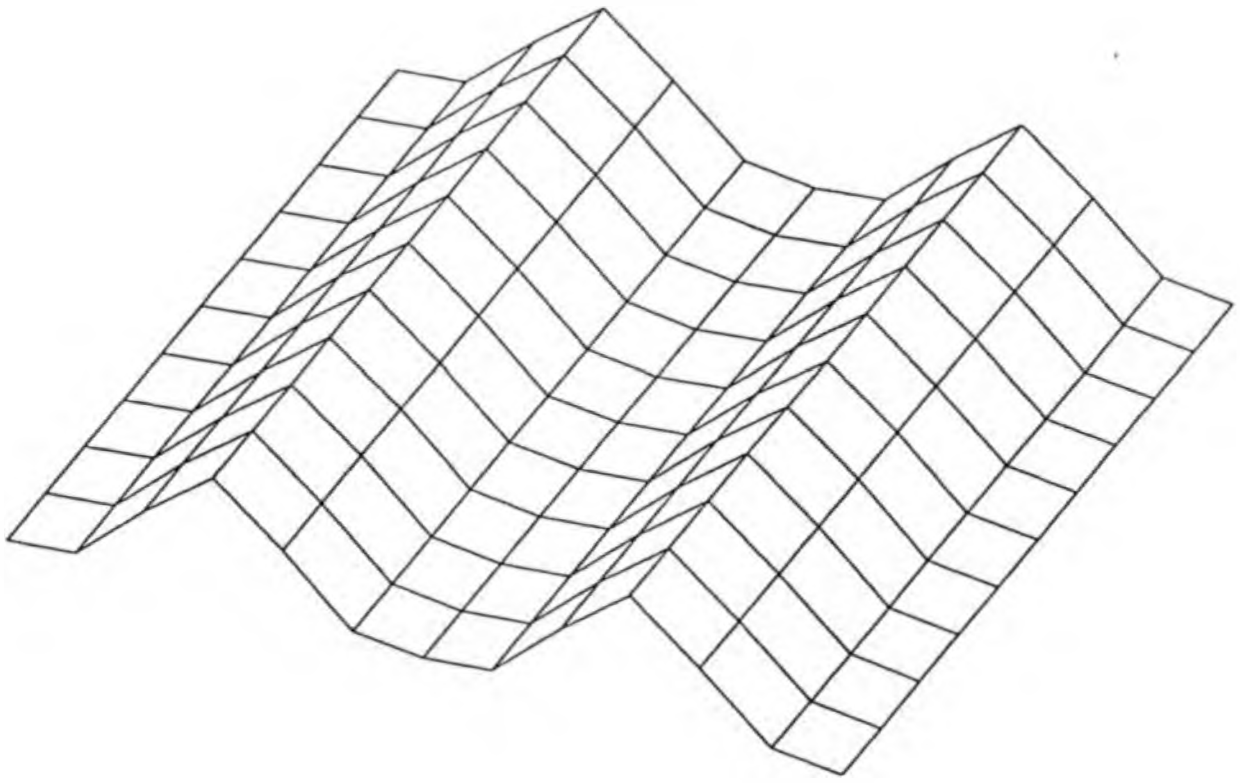


Figure C.1 (a) Unit-step tracing

```
ugtrace -us -dm -fo example.c1.out < example.c1
ugdisp -dm -ed 2.5 4 -5 < example.out
```

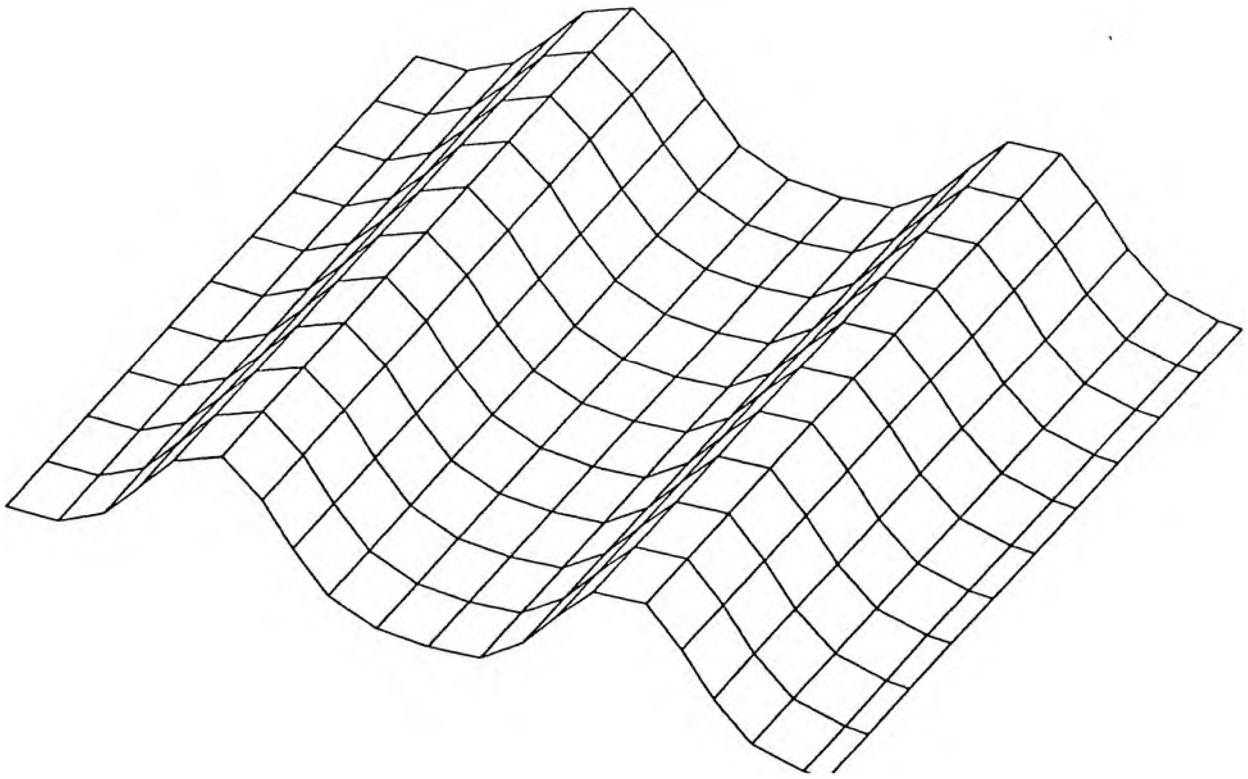


Figure C.1 (b) Unit-length tracing

```
ugtrace -ul -dm -fo example.cl.out < example.cl
ugdisp -dm -ed 2.5 4 -5 < example.out
```

```
{
  Example C.2  Varying Tracing Magnitude
  and Using Multiple Domains
}

par s, t
var x, y, z

x = t
y = tan(t)
z = s

dom 0 <= s <= 5, 1

const pi = 3.141592654

{ setting up constants for defining multiple domains }

const R1 =     pi / 10
const R2 = 2 * pi / 10
const R3 = 3 * pi / 8

const R4 = pi - R3
const R5 = pi - R2
const R6 = pi - R1

const R7 = R1 - pi
const R8 = R2 - pi
const R9 = R3 - pi

{ Use this block rough plotting }
{
dom -pi <= t <= R7, 0.10   { plot from -pi to -pi/2 }
dom  R7 <= t <= R8, 0.20
dom  R8 <= t <= R9, 0.25

dom -R3 <= t <= -R2, 0.25  { plot from -pi/2 to pi/2 }
dom -R2 <= t <= -R1, 0.20
dom -R1 <= t <=  R1, 0.10
dom  R1 <= t <=  R2, 0.20
dom  R2 <= t <=  R3, 0.25

dom R4 <= t <= R5, 0.10    { plot from pi/2 to pi }
dom R5 <= t <= R6, 0.20
dom R6 <= t <= pi, 0.25
}
{ Use this block detail plotting }
```

```
dom -pi <= t <= R7, 0.05    { plot from -pi to -pi/2 }
dom  R7 <= t <= R8, 0.10
dom  R8 <= t <= R9, 0.15

dom -R3 <= t <= -R2, 0.15  { plot from -pi/2 to pi/2 }
dom -R2 <= t <= -R1, 0.10
dom -R1 <= t <=  R1, 0.05
dom  R1 <= t <=  R2, 0.10
dom  R2 <= t <=  R3, 0.15

dom R4 <= t <= R5, 0.05    { plot from pi/2 to pi }
dom R5 <= t <= R6, 0.10
dom R6 <= t <= pi, 0.15

{
  Comment:

  As t approaches multiples of half pi, rate of change
  of tan(t) increases.  Therefore, decreasing the step
  size of t stablizes the distance between the points
  traced.  Changing of step size is done by splitting
  the domain into smaller intervals where each interval
  has its step size.
}
```

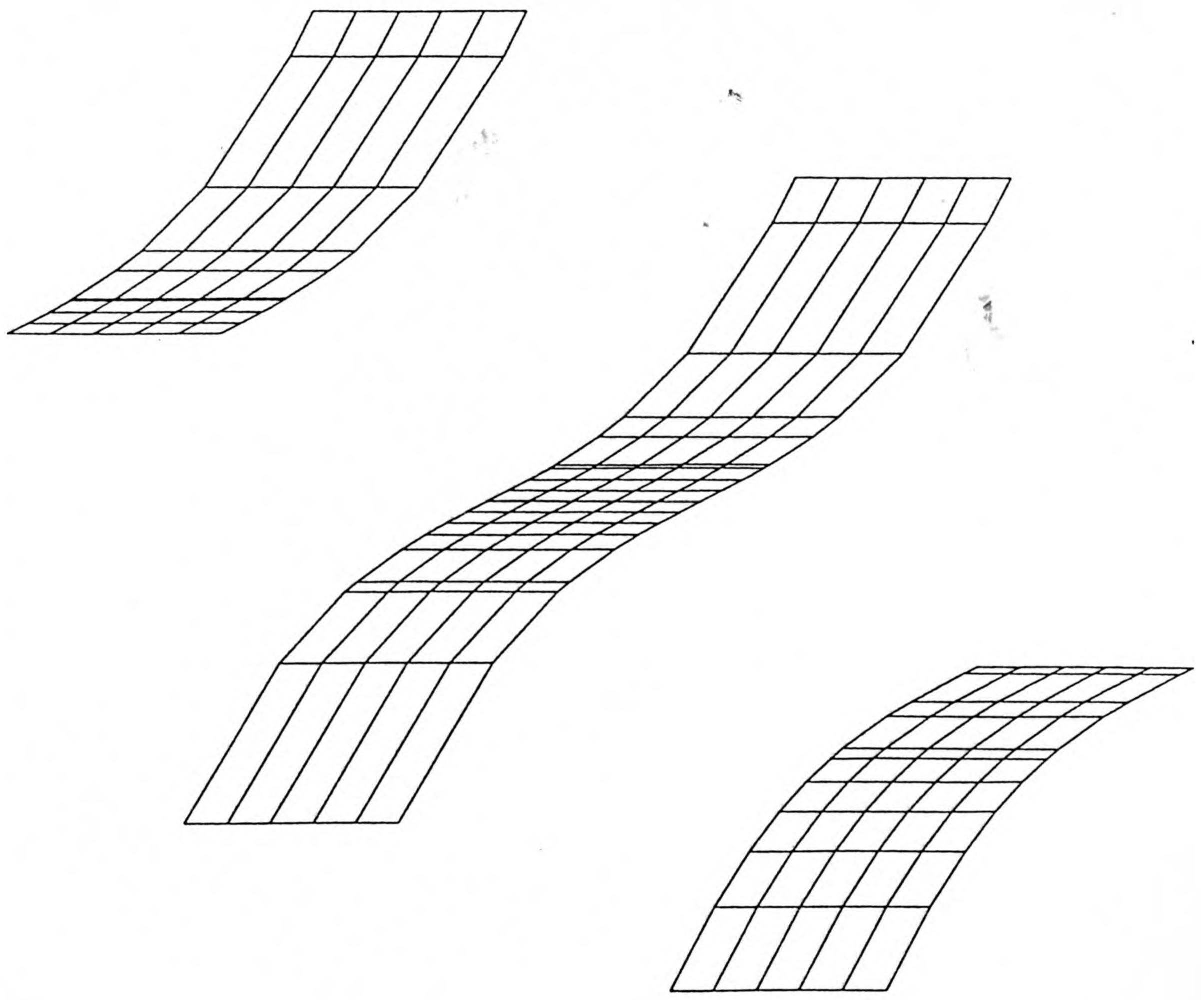


Figure C.2 (a) Low Resolution

```
ugtrace -us -dm -fo example.c2.out < example.c2
ugdisp -dm -ed 0.75 0.25 -3 < example.c2.out
```

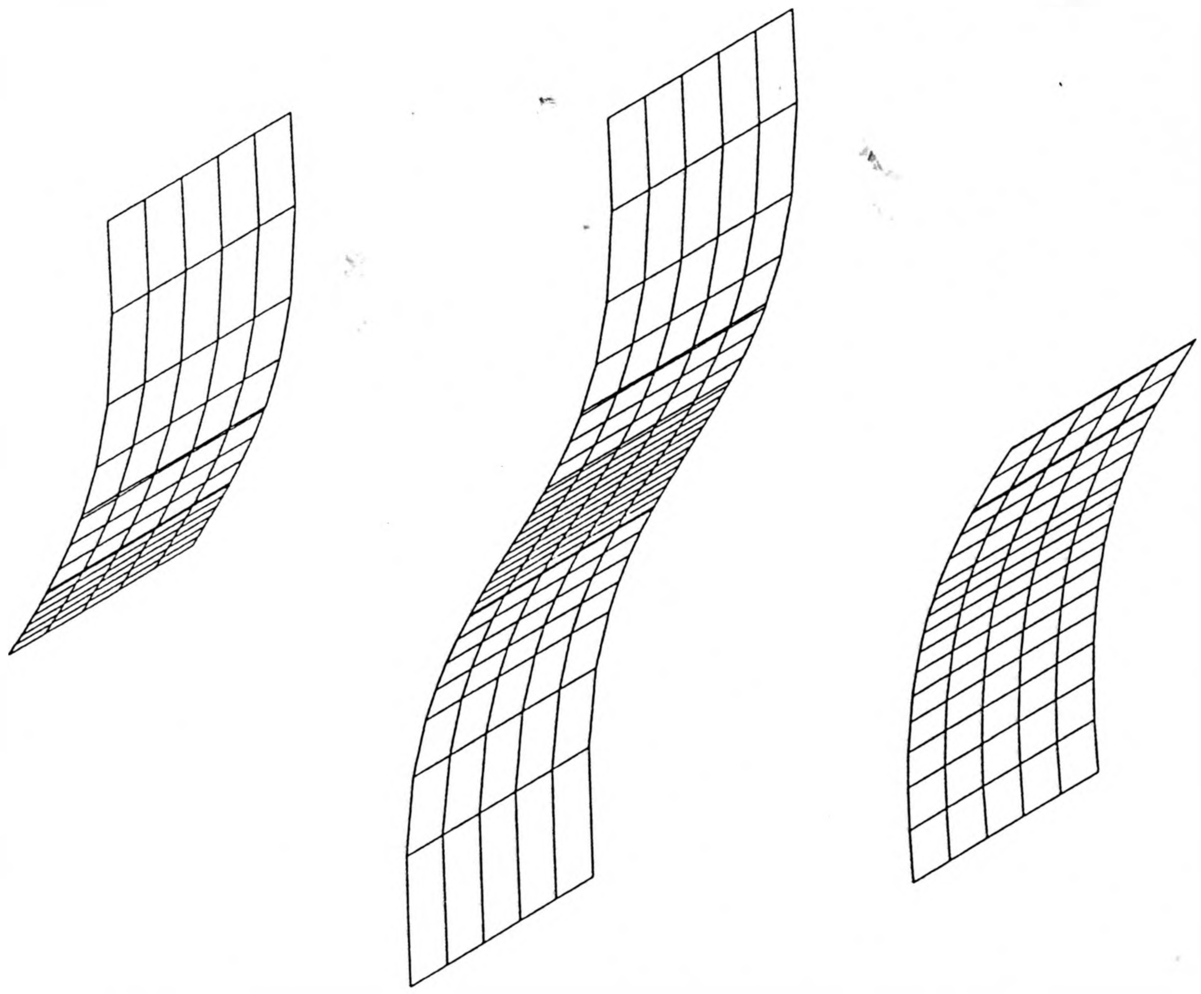


Figure C.2 (b) High Resolution

```
ugtrace -us -dm -fo example.c2.out < example.c2
ugdisp -dm -ed 0.75 0.25 -3 < example.c2.out
```

```
{
  Example C.3  Mesh Display Versus Patch Display

  This example demonstrates the difference
  between the two displaying modes
}

par u, v
var x, y, z

x = sin(v) * cos(u)
y = sin(v) * sin(u)
z = cos(v)

const  pi = 3.141592654

const  Lu = 0
const  Ru = 2 * pi
const  du = pi / 6

const  Lv = -pi
const  Rv =  pi
const  dv = pi / 6

dom Lu <= u <= Ru, du
dom Lv <= v <= Rv, dv
```

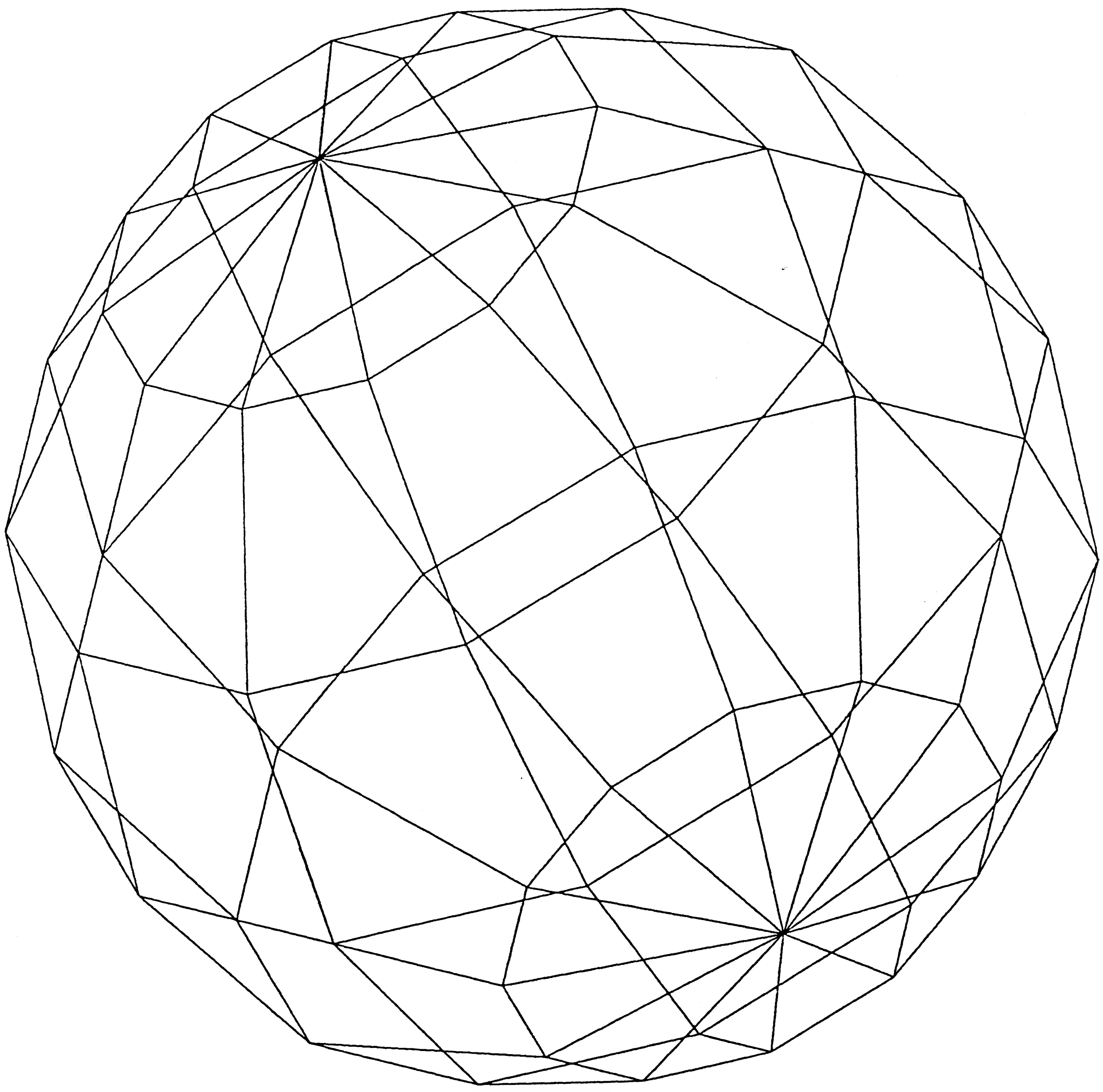


Figure C.3 (a) Mesh display

```
ugtrace -us -dm -fo example.c3.out < example.c3
ugdisp -dm -ed 1 1 -1 < example.c3.out
```

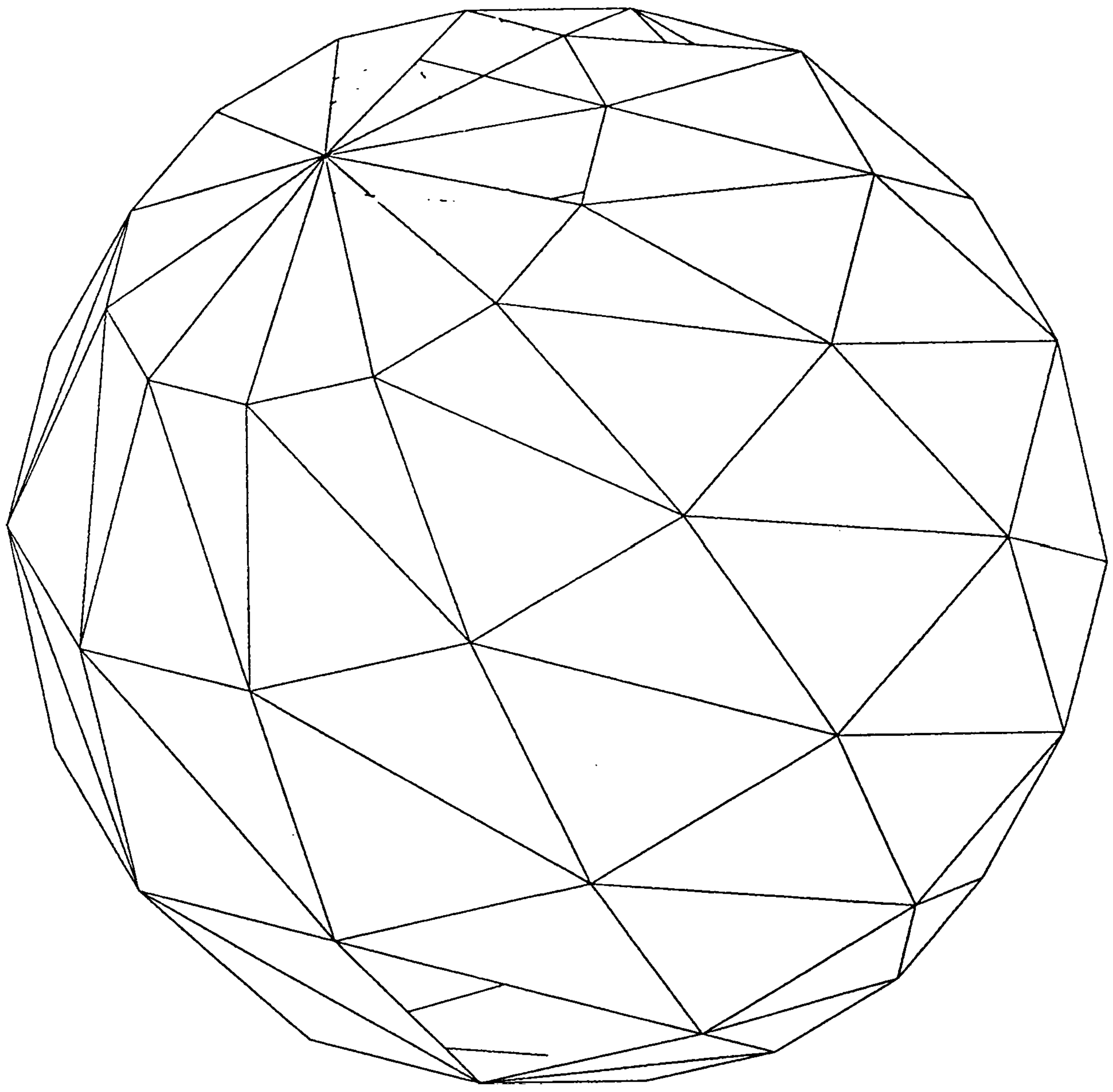


Figure C.3 (b) Patch display

```
ugtrace -us -dm -fo example.c3.out < example.c3
ugdisp -dm -ho -ed 1 1 -1 < example.c3.out
```

Appendix D

ALGORITHMS AND DATA STRUCTURES FOR UGTRACE

D.1 Overview

D.1.1 CONTROL FLOW

Main control flow is reflected in the leading module main(). There are 5 processing phases. Each phase is initiated by one or more leading sub-modules in main(). The control flow is shown as follow:

Phase 1 - Scanning Command Line [Input Section]
Command line is scanned for run-time options. Getcmdln() sets various switches for run-time options. Chkcmdln() checks for error conditions and logical conflicts on switches set. If the log option is set, then a logfile is created with a system generated filename.

Phase 2 - Loading Specification [Input Section]
Getspec() reads the specification from the specified source input to a memory file. Upon successful completion, getspec() returns the number of lines it has read to the memory file. Comments in the specification is also replaced with blanks by getspec(). If getspec() cannot open the file or if the file is empty or too large, then the program will be aborted without returning to main().

Phase 3 - Parsing Specification [Input Section]
Intrspec() is a leading sub-module to the parsing process. The memory file is parsed line by line. The specification is transformed from text form to internal data structures. Information records and expression trees are built during this parsing process. Following intrspec(), chkspec() and audit() are called to perform checking and auditing functions on the resulting data structures. Missing information is detected and default values are applied at this stage.

Phase 4 - Tracing [Tracing Section]
Plotspec() is the leading sub-module to the tracing process. It takes the expression trees and various information records and plots some points in the given domains. Upon successful

completion, plotspec() returns the number of points plotted and a hierarchy of point records.

Phase 5 - Exporting to UNIGRAFIX format [Output Section]
Export() is the leading sub-module to the exporting process. The hierarchy of records generated from plotspec() is transformed to the UNIGRAFIX format. Each point is given a unique vertex name. Vertices of a curve or a patch are then referenced using the vertex names. Curves and patches are also given unique names for identifying purposes. Two files are used, one to store the vertices and the other to store the curves or patches. When all the records in the hierarchy structure are visited, the second file is appended to the first to produce one output file.

D.1.2 SOURCE FILES

UGTRACE has 11 files of C source code:

```
ugtrace.c  -- leading module: main()

cmdln.c    -- miscellaneous functions for parsing command line:
              getcmdln(), chkcmdln()

input.c    -- miscellaneous functions for getting input:
              getspec(), intrspec(), getarg()

parse.c    -- miscellaneous functions for parsing input:
              getconst(),   getvar(),      getpar(),   getdom(),
              geteqt(),     getoken(),     chkspec(),  namegood(),
              nameused(),   isconstant(),  isparameter(),
              isvariable(), isdomain(),    isletter(), findulen()

expr.c     -- miscellaneous functions for parsing expressions:
              isform(),  isexpr(),  isterm(),   isfact1(),
              isfact2(), isargu(),  isiden(),   islit(),
              isfunc(),  isaddop(), ismultop(), ispowop()

evalu.c    -- miscellaneous functions for evaluating expression:
              evalu(), evalu2()

unitstep.c -- miscellaneous functions for unit step tracing:
              plotspec(),   plotsurfaces(), plot1surface(),
              plotcurves(), plot1curve(),   mkpoint(),
              mkcurve(),    mksurface()

unitlen.c  -- miscellaneous functions for unit length tracing:
              unit1surface(), unit1curve(), homein()

export.c   -- miscellaneous functions for exporting to UNIGRAFIX
              format: export(), putsurface(), putcurve(),
               putlimit(), putaxis(), putcmap(), nextid()

message.c   -- miscellaneous functions for displaying messages:
               message()

audit.c     -- miscellaneous functions for auditing:
               audit(), logsurface(), logcurve(), logpoint(),
               prtexpr(), prtop()
```

All the functions are listed in alphabetical order at the end of this appendix.

D.2 Input section

D.2.1 SCANNING COMMAND LINE [PHASE 1]

The command line arguments are stored in an array of strings, one per string. The pointer to this array, called argv, and the number of arguments the program was invoked with, called argc, are passed to main() when it begins executing. Argc and argv are then passed to getcmdln().

Getcmdln() looks at the argument strings sequentially, and compares them with the valid switches set. If a match is found, then the flag for that switch is raised. A flag is an integer whose value is either 0 or 1. For switches like "-fi" and "-fo", the next argument is interpreted as a filename instead. If no match is found for the argument string, then the error flag for unresolved switch is raised.

Algorithm for getcmdln():

```
1  Set all switches to their default values

2  If no more argument, then goto 11

3  If match "-fi" switch, then
   3.1  if next argument is a valid filename, then
          - copy next argument to input filename
          - skip next argument
        else raise "fi" error flag
   3.2  goto 2

4  If match "-fo" switch, then
   4.1  if next argument is a valid filename, then
          - copy next argument to output filename
          - skip next argument
        else raise "fo" error flag
   4.2  goto 2

5  If match "-us" switch, then
   5.1  turn unit-length tracing mode off
   5.2  goto 2

6  If match "-ul" switch, then
   6.1  turn unit-length tracing mode on
   6.2  goto 2

7  If match "-dp" switch, then
   7.1  turn mesh display mode off
   7.2  goto 2

8  If match "-dm" switch, then
   8.1  turn mesh display mode on
   8.2  goto 2

9  If match "-a" switch, then
   9.1  turn axis mode on
   9.2  goto 2

10 If match "-l" switch, then
   10.1 turn log mode on
   10.2 goto 2

11 Raise unresolved switch error flag

12 Return
```

Note that it is possible in the above algorithm to set a switch more than once. If a switch is set more than once, then the previous setting is overwritten by the new one.

Chkcmdln() is called after getcmdln() to check the status of various switches.

```
Algorithm for chkcmdln():

1  Check status of all switches and report their settings

2  Check whether input path is accessible

3  Check whether output path is accessible

4  Return
```

Any checking and validating of command line arguments should be included in chkcmdln().

D.2.2 LOADING SPECIFICATION [PHASE 2]

After the scanning of command line arguments, the plot specification is read from the input stream. Getspec() takes the input filename, loads the complete file to a temporary memory buffer and returns the number of lines of the file. Comments in the file are also substituted with white spaces.

Algorithm for getspec():

```
1  Allocate memory buffer for an array of lines

2  Open input stream

3  While (not EOF) and (buffer not full) do
   3.1  read one line of text to buffer
   3.2  increment line counter by 1

4  Close input stream

5  Filter out comments

6  Return total number of lines
```

Nesting of comments is allowed. A counter is used to count the level of comments. The counter is initially set to 0. The buffer is examined character by character. If a "{" is encountered, then the counter is increased by 1. On the other hand, if a "}" is encountered, the counter is decreased by 1. When the counter has a positive value, the characters examined are substituted by white spaces. An exception is the newline characters. They are not replaced even if the counter has a value larger than zero. An error occurs whenever the counter becomes negative, or when all the characters are examined and the counter does not equal to 0.

D.2.3 PARSING SPECIFICATION [PHASE 3]

D.2.3.1 Parsing Shell

Parsing of a line is performed according to the category of information the line contains. Intrspec() is a shell which looks at the buffer of lines from phase 2 and determines the required parsing module for each line. Each line from the buffer, when processes by intrspec, is split into an array of words.

Algorithm for intrspec():

```
1  Allocate memory for an array of words

2  While there is a line to process do
```

```
    2.1  Split line into an array of words
    2.2  If first word indicates CONSTANT statement then
           parse a CONSTANT statement
    2.3  If first word indicates PARAMETER statement then
           parse a PARAMETER statement
    2.4  If first word indicates VARIABLE statement then
           parse a VARIABLE statement
    2.5  If first word indicates DOMAIN statement then
           parse a DOMAIN statement
    2.6  Otherwise parse an equation

3  Release memory space for word array

4  Release memory space for line array

5  Return
```

A word is alphanumeric string of characters terminated with a white space or a newline character. The string cannot contain any of the special characters: "+", "-", "*", "/", "^", "(", ")", "<", ">", "=", and ",". Each special character, except "<" and ">", stands alone as a word and does not require white space or newline delimiters. "<" or ">" may be followed by "=" to form one word "<=" or ">=". Each word when copied from the line buffer to the word array is stripped of leading and trailing blanks, and appended with a null character " ". The end of the word array is marked by a null string "" which consists of a single null character.

Note that "_" and "." are not special characters and can be used as part of a word.

D.2.3.1.1 Parsing a CONSTANT Statement

Called by intrspec(), getconst() takes an array of words and builds an internal record for a constant identifier. The record is of type identifier:

```
struct identifier
{
  int    itype;           /* type CONSTANT       */
  char   nam[WORDSIZ];    /* constant name       */
  double valu;            /* constant value      */
  double vmin, vmax;      /* not used            */
  struct expr *f;         /* expression, if any */
}
```

Type identifier is also used by parameter identifiers and variable identifiers. The integer value of field itype determines the content of the record. Field nam store the character string which denotes the constant. Field valu store the numeric value of the constant. If an expression is used to define the value, then the expression is referred to by pointer field f. Expressions are discussed in D.2.3.7.

An array of pointers,

```
        struct identifier *const[NUMCONST]
```

are used to keep track of the constant records. NUMCONST is the size of the array. For each successful invocation of getconst(), a new constant record is added to *const[].

```
        Algorithm for getconst():

        1  Verify that the second word in the array is an
             identifier

        2  Verify that the identifier is not previously declared
             as constant, parameter or variable

        3  Verify that *const[] is not full yet

        4  Verify that the third word in the array is an equal
             sign, "="

        5  Create an expression from the rest of the words

        6  Create an identifier record

        7  7.1  set field itype to CONSTANT
           7.2  copy the second word to field nam
           7.3  point field f to expression created
           7.4  evalu expression and store value to field valu

        8  Return
```

Note that the above algorithm does not allow a constant to be redeclared or redefined.

D.2.3.1.2 Parsing a PARAMETER Statement

Called by intrspec(), getpar() takes an array of words and builds an internal record for a parameter identifier. The record is of type identifier:

```
        struct identifier
        {
          int    itype;          /* type PARAMETER            */
          char   nam[WORDSIZ];   /* parameter name            */
          double valu;           /* instantaneous par. value */
          double vmin;           /* not used                  */
```

```
    double vmax;          /* number of domains          */
    struct expr *f;       /* not used                   */
  }
```

Field itype is set to PARAMETER. Field nam stores the character string which denotes the parameter. Field valu and field vmax are not used in declaration. Field valu is used to store the instantaneous value of the parameter during the tracing. Field vmax is used to count the number of domain ranges.

An array of pointers,

```
        struct identifier *param[NUMPARAM]
```

are used to keep track of the parameter records. NUMPARAM is the size of the array. For each successful invocation of getpar(), a new parameter record is added to *param[]. The number of parameter records allowed is arbitrary and not necessarily two.

```
Algorithm for getpar():

1  Verify that the second word in the array is an
     identifier

2  Verify that the identifier is not previously declared
     as constant, parameter or variable

3  Verify that *param[] is not full yet

4  Create an identifier record

5  5.1  set field itype to PARAMETER
   5.2  copy the identifier to field nam
   5.3  Create an domain array (see D.2.3.5)

6  If next word equal ",", then goto step 1
   else if next word not equal "" then error

7  Return
```

Note that the above algorithm does not allow a parameter to be redeclared.

Each parameter record may have more than one domain interval. The information about domains are stored in another array called dom. The number of elements in dom is stored in field vmax of the parameter record. Domains are discussed in section D.2.3.5.

D.2.3.1.3 Parsing a VARIABLE Statement

Called by intrspec(), getvar() takes an array of words and builds an internal record for a variable identifier. The record is of type identifier:

```
        struct identifier
        {
          int    itype;          /* type VARIABLE             */
          char   nam[WORDSIZ];   /* variable name             */
          double valu;           /* instantaneous var. value  */
          double vmin;           /* min recorded value        */
          double vmax;           /* max recorded value        */
          struct expr *f;        /* expression for variable   */
        }
```

Field itype is set to VARIABLE. Field nam stores the character string which denotes the variable. Field valu, field vmin, field vmax and field f are not used in declaration. Field valu is used to store the instantaneous value of the variable during the tracing. Field vmin and field vmax are used to stored the minimum and maximum value of the variable. Field f is the expression used to evaluate the instantaneous value of the variable.

An array of pointers,

```
        struct identifier *var[NUMVAR]
```

are used to keep track of the parameter records. NUMVAR is the size of the array. For each successful invocation of getvar(), a new variable record is added to *var[]. The number of variable records allowed is arbitrary and not necessarily three.

```
        Algorithm for getvar():

        1  Verify that the second word in the array is an
             identifier

        2  Verify that the identifier is not previously declared
             as constant, parameter or variable

        3  Verify that *var[] is not full yet

        4  Create an identifier record

        5  5.1  set field itype to VARIABLE
           5.2  copy the identifier to field nam

        6  If next word equal ",", then goto step 1
           else if next word not equal "" then error

        7  Return
```

Note that the above algorithm does not allow a variable to be redeclared.

D.2.3.1.4 Parsing a DOMAIN Statement

Called by intrspec(), getdom() takes an array of words and builds an internal record for a domain interval. The record is of type domain:

```
        struct domain
        {
          int    flag;      /* inequality flag code           */
          double dmin;      /* lower limit of domain interval */
          double dmax;      /* upper limit of domain interval */
          double trace;     /* trace magnitude                */
          double precision; /* unit length trace precision    */
        }
```

Field flag is used to indicate interval type. There are 4 types of intervals:

```
    Type 1:  ( dmin , dmax )          Type 3:  ( dmin , dmax ]
    Type 2:  [ dmin , dmax ]          Type 4:  [ dmin , dmax )
```

where "(" and ")" denote an open end of a interval, "[" and "]" denote a closed end of a interval. Field dmin and field dmax store the lower and upper limit of the domain interval. Field trace stores the trace magnitude. Field precision is used only in unit length mode.

An array,

```
        struct domain (*dom[NUMPARAM])[NUMDOM]
```

is used to store domain information. Each member of the array is a pointer to an array of domain record. The size of the array of pointers is NUMPARAM. When a parameter record is created and referred to by a slot in "param", an array of domain records is created as well. The array index used in "param" is also used in "dom" to index the array of domain record for that parameter.

NUMDOM is the maximum number of intervals allowed for each parameter. For each successful invocation of getdom(), a new domain record is initialized.

```
        Algorithm for getdom():

        1  Verify that the second word in the array is a number or
             a constant

        2  Verify that the next word is an inequality sign, "<",
             ">", "<=" or ">="
```

```
 3  Verify that the next word is a declared parameter
      identifier

 4  Verify that the next word is an inequality sign, "<",
      ">", "<=" or ">="

 5  Verify that the next word is a number or a constant

 6  Verify that the two signs are not contradicting, e.g.
      "<" and ">"

 7  If next word is ",", then
      get a number or constant for trace magnitude;
    else if tracing mode is unit step then
      apply default for trace magnitude: 1% of interval
    else
      left trace magnitude blank

 8  If next word is ",", then
      get a number or constant for trace precision;
    else
      left trace magnitude blank

 9  9.1  Copy all the values to correct parameter domain
            array slot
    9.2  Increase the vmax field in the parameter record
            by 1

10  Return
```

For unit length tracing, default trace magnitude and trace precision cannot be determined until all the variables are defined. There are more checks on the integrity of the domain intervals in chkspec().

Note that the above algorithm cannot detect overlapping of intervals. The intervals remain in the order that they are presented in the specification.

D.2.3.1.5 Parsing an Equation

Called by intrspec(), geteqt() takes an array of words and scans for an equation. There is no internal record for an equation. The expression on the right hand side of an equation is represented by a binary tree structure. Field f in a variable record is the pointer to the root of an expression tree. Expressions are discussed in section D.2.3.7.

Algorithm for geteqt():

```
1  Verify that the second word in the array is a declared
```

variable

2 Verify that the next word is an equal sign "="

3 Create an expression from the rest of the words

4 Verify that field f of the variable record is not used

5 Assign field f to the pointer value to new expression

6 Return

D.2.3.1.6 Parsing an Expression

An expression is represented by a binary tree structure. Each node is of type expr:

```
struct expr
{
  int ctype;
  union cell info;
  struct expr *l, *r;
}
```

Each node of either an operator or an operand. Field ctype is used to indicate the content of field cell. Valid values of ctype is defined by the following constants:

```
#define INTEGER   0x01
#define REAL      0x02
#define OPERATOR  0x08
#define FUNCTION  0x10
#define CONSTID   0x20
#define PARAMID   0x40
#define VARID     0x80
```

Field info is a variant record which can be an integer, a double, a string, or a pointer:

```
union cell
{
  int ival;
  double rval;
  struct identifier *ptr;
  char sval[WORDSIZ];
}
```

The process of parsing an array of words to build an expression tree consist of a set of recursive functions. The functions resemble the various level of the grammar which defines an expression.

The grammar:

```
<expr>   ::= <term> | <term><addop><expr>
<term>   ::= <fact1> | <fact1><multop><term>
<fact1>  ::= <func><fact2> | <fact2>
<fact2>  ::= <argu> | <argu><powop><fact2>
<argu>   ::= <lit> | <iden> | "("<expr>")"
<iden>   ::= <string>
<lit>    ::= <real> | <int>
<addop>  ::= "+" | "-"
<multop>::= "*" | "/"
<func>   ::=  "sin"  |  "cos"  |  "tan"  |
             "asin"  | "acos"  | "atan"  |
              "sinh" |  "cosh" |  "tanh" |
             "asinh" | "acosh" | "atanh" |
              "exp"  |  "log"  | "log10" | "sqrt"
<powop>  ::= "^"
```

The functions:

```
isform()   -- lead module
isexpr()   -- check for an <expr>
isterm()   -- check for a <term>
isfact1()  -- check for a <fact1>
isfact2()  -- check for a <fact2>
isargu()   -- check for a <argu>
isiden()   -- check for an <iden>
islit()    -- check for a <lit>
isfunc()   -- check for a <func>
isaddop()  -- check for an <addop>
ismultop() -- check for a <multop>
ispowop()  -- check for a <powop>
```

All the functions share a global pointer, "bufp" which points to the word currently being processed. If the word is used, "bufp" is advanced to the next word. If the array of words is not syntactically correct, then an error exit will occur at the level the error is detected. On successful return, each function returns a node of type expr.

The algorithms of the functions are described as follow:

Algorithm for isform():

```
1  1.1 Set "bufp"  to the first word of the given array;
   1.2 Stop if no first word is found

2  2.1 Call isexpr() to check for an <expr>
   2.2 If yes, return the tree structure isexpr() returned

3  Otherwise return NULL
```

Algorithm for isexpr():

```
1  Return NULL if no word is found

2  2.1 Call isterm() to check for a <term>
   2.2 If no, return NULL

3  3.1 Call isaddop() to check for a <addop>
   3.2 If no, goto step 6

4  4.1 Call isterm() to check for another <term>
   4.2 If no, error exit

5  5.1 Append the two <term> subtree as left and right
       child of the <addop> node
   5.2 Repeat step 3

6  Return the <expr> subtree build
```

Algorithm for isterm():

```
1  Return NULL if no word is found

2  2.1 Call isfact1() to check for a <fact1>
   2.2 If no, return NULL

3  3.1 Call ismultop() to check for a <multop>
   3.2 If no, goto step 6

4  4.1 Call isfact1() to check for another <fact1>
   4.2 If no, error exit

5  5.1 Append the two <fact1> subtrees as left and right
       children of the <multop> node
   5.2 Repeat step 3

6  Return the <term> subtree build
```

Algorithm for isfact1():

```
1  Return NULL if no word is found

2  2.1 Call isfunc() to check for a <func>
   2.2 If yes, call isfact2() to check for a <fact2)
       2.2.1 If yes, goto step 4
       2.2.2 If no, error exit

3  3.1 Call isfact2() to check for a <fact2)
   3.2 If yes, goto step 4
   3.2 If no, return NULL
```

```
6  Return the <fact1> subtree build
```

Algorithm for isfact2():

```
1  Return NULL if no word is found

2  2.1 Call isargu() to check for an <argu>
   2.2 If no, return NULL

3  3.1 Call ispowop() to check for a <powop>
   3.2 If no, goto step 6

4  4.1 Call isargu() to check for another <argu>
   4.2 If no, error exit

5  5.1 Append the two <argu> subtree as left and right
       child of the <powop> node
   5.2 Repeat step 3

6  Return the <fact2> subtree build
```

Algorithm for isargu():

```
1  Return NULL if no word is found

2  2.1 Call islit() to check for a <lit>
   2.2 If yes, goto step 5

3  3.1 Call isiden() to check for an <iden>
   3.2 If yes, goto step 5

4  If "bufp" points to a "(", then
   4.1 Advance "bufp" to point at the next word
   4.2 Call isexpr() to check for an <expr>
       4.2.1 If no, error exit
       4.2.2 If yes, then
               if "bufp" points to a ")",
               then goto step 5
               else error exit

5  Return the <argu> subtree build
```

Algorithm for isiden():

```
1  Return NULL if no word is found

2  If "bufp" points to "-",
   then sign = -1 and advance "bufp"

3  Return NULL if no word is found
```

```
4  If "bufp" points to declared constant or
     parameter identifier,
   then create and initialize a node

5  Return the <iden> subtree build
```

Algorithm for islit():

```
1  Return NULL if no word is found

2  If "bufp" points to "-",
   then sign = -1 and advance "bufp"

3  Return NULL if no word is found

4  If "bufp" points to an integer value or a real number,
   then create and initialize a node

5  Return the <lit> subtree build
```

Algorithm for isfunc():

```
1  Return NULL if no word is found

2  If "bufp" points to a word which matches
     any of the defined mathematical functions
   then create and initialize a node

3  Return the <func> subtree build
```

Algorithm for isaddop():

```
1  Return NULL if no word is found

2  If "bufp" points to a "+" or "-"
   then create and initialize a node

3  Return the <addop> subtree build
```

Algorithm for ismultop():

```
1  Return NULL if no word is found

2  If "bufp" points to a "*" or "/"
   then create and initialize a node

3  Return the <multop> subtree build
```

Algorithm for ispowop():

```
1  Return NULL if no word is found
```

```
2  If "bufp" points to a "^"
   then create and initialize a node

3  Return the <powop> subtree build
```

D.2.3.2 Overall Checking

The focus of the parsing shell is on the syntax of text. Missing or contradicting informations are not detected. Chkspec() applies series of checks to detect error situations getspec() failed to trap. Default tracing values for unit length tracing is determined in chkspec() as well.

Algorithm for chkspec():

```
1  Detect and drop constants which have been declared
     but not used

2  2.1  Verify that at least 1 parameter has been declared
   2.2  Detect and drop parameters which have been
          declared but not used in any equation
   2.3  Verify that no more than 2 parameter are declared
   2.4  Verify that for all parameters which are declared
          and used, at least one domain is associated
          with the parameter

3  3.1  Verify that at least 1 variable has been declared
   3.2  Detect and drop variables which have been declared
          but not used in any equation
   3.3  Verify that no more than 3 variables are declared
   3.4  Verify that for all variables which are declared
          and used, some expression is defined for the
          variable

4  For every domain record, verify that
   4.1  the trace magnitude for that interval is not
          negative; otherwise multiply by -1
   4.2  the precision value for that interval is not
          negative; otherwise multiply by -1

   4.3  the trace magnitude for that interval is non-zero;
          otherwise if tracing mode is unit-step, then
          change magnitude to default: 1% of (upper limit
          - lower limit)

5  For every domain record, adjust lower and upper limits
   5.1  Find delta = 0.1% of (upper limit - lower limit)
   5.2  If upper end is open, then
          decrease upper limit by delta
   5.3  If lower end is open, then
          increase lower limit by delta
```

```
6  If tracing mode is unit-length, then
     for every domain record,
     if tracing magnitude is zero, then
     (Curve)
     6.1.1  Find a point when parameter equals lower limit
     6.1.2  Increment parameter by 1% of interval
     6.1.3  Find second point
     6.1.4  Set tracing magnitude to distance between 2
              points
     (Surface)
     (If domain record specifies an interval of
      parameter1)
     6.2.1  Set value of parameter2 to lower limit of
              first domain interval of parameter2
     6.2.2  Find a point when parameter1 equals lower
              limit
     6.2.3  Increment parameter1 by 1% of interval
     6.2.4  Find second point
     6.2.5  Set tracing magnitude to distance between 2
            points
     (If domain record specifies an interval of
      parameter2)
     6.3.1  Set value of parameter1 to lower limit of
              first domain interval of parameter1
     6.3.2  Find a point when parameter2 equals lower
              limit
     6.3.3  Increment parameter2 by 1% of interval
     6.3.4  Find second point
     6.3.5  Set tracing magnitude to distance between 2
            points
```

A successful completion of the input section should have at least 1 parameter record and 2 variable records created. Each parameter record has at least 1 domain record associated. Each variable record has a valid expression tree built.

D.3 Tracing section

D.3.1 TRACING [PHASE 4]

In the tracing section, parameter values are advanced according to the tracing magnitude. The expression trees are then evaluated to yield the corresponding variable values. There are two modes, unit-step tracing and unit-length tracing. Unit-step routines are "us" prefixed, while unit-length routines are "ul" prefixed.

The output of this phase is an hierarchy structure of coordinates records. Each combination of domains is represented by a surface node of type:

```
        struct surface
        {
          int sflag;            /* error flag                   */
          int i1;               /* domain index to 1st parameter */
          int i2;               /* domain index to 2nd parameter */
          double nump;          /* # of points in this patch    */
          double numc;          /* # of curves in this patch    */
          struct curve *root;   /* pointer to first curve node  */
          struct surface *ns;   /* pointer to next surface node */
        }
```

Each surface consists of a collection of curves. The first curve is pointed to by field root. Curve nodes are the next level of records:

```
        struct curve
        {
          int cflag;            /* error flag                  */
          int i1;               /* index to parameter          */
          int i2;               /* index to domain             */
          double nump;          /* # of points in this curve   */
          struct point *head;   /* pointer to first point node */
          struct curve *nc;     /* pointer to next curve node  */
        }
```

Each curve consists of a collection of points. The first point is pointed to by field head. Point nodes are the third level of records:

```
        struct point
        {
          int pflag;          /* error flag                  */
          double u[2];        /* parameter values            */
          double x[2];        /* variable values             */
          struct point *np;   /* pointer to next curve node */
        }
```

Field x is an array of real numbers which stores the coordinates of a point. Field u is an array of real numbers which stores the corresponding parameter values.

The hierarchy does not need to start with a surface node. If the geometric form traced is a curve, then the hierarchy begins at the curve node level. Either one of the three global variables "cur2d", "cur3d" and "sur3d" is used to represent the curve or surface being trace. Prefix "cur" means curve and prefix "sur" means surface. Suffix "2d" means 2 dimensions and suffix "3d" means 3 dimensions. "Cur2d" and "cur3d" are pointers to a curve node, while "sur3d" is a pointer to a surface node.

Plotspec() is the leading module to the tracing section. It takes all the parameter and variable records, initiates the required plotting shell, and returns the number of points plotted.

Algorithm for plotspec():

```
1  If number of variables equals to 2, then
     plot a curve in 2 dimensional space
   else if number of parameters equals to 1, then
     plot a curve in 3 dimensional space
   else
     plot a surface in 3 dimensional space

2  Count number of points plotted

3  Return number of points plotted
```

Plotcurves() is the lead module of the plotting shell for curves. An index to the parameter it should trace is passed as an argument. It then traces a curve for each domain interval of the indexed parameter.

Algorithm for plotcurves():

```
1  While there is more domains do
   if unit-length mode is mode, then
     plot 1 curve in unit length mode
   else
     plot 1 curve in unit step mode

2  Return sequence of curve nodes
```

The surface plotting shell is similar to the curve plotting shell. However, there is an extra loop to control the domain index of the second parameter. Plotsurfaces() is the lead module to the surface plotting shell.

Algorithm for plotsurfaces():

```
1  While there is more first parameter domains do
   while there is more second parameter domains do
     if unit-length mode is mode, then
       plot 1 surface in unit length mode
     else
       plot 1 surface in unit step mode

2  Return sequence of surface nodes
```

D.3.1.1 Unit-Step Tracing

Uslcurve() unit-step traces a curve on the parameter and the domain requested. Indices to the parameter and the domain it should trace are passed as arguments.

Algorithm for uslcurve():

```
1  Create a new curve node, error exit if failure

2  Set current value of indexed parameter to lower limit
     of indexed domain

3  3.1 evaluate the variables
   3.2 create a point record
   3.3 append to the point nodes of current curve

4  Increment parameter value by trace magnitude

5  If parameter value >= upper limit, then
   5.1 set quit flag
   5.2 if parameter value > upper limit, then
         set to upper limit

6  If quit flag is raised, then
     goto step 7
   else
     goto step 3

7  Return current curve node
```

The philosophy of uslsurface() is similar to uslcurve().

Algorithm for uslsurface():

```
1  Create a new surface node, error exit if failure

2  Set current value of second parameter to lower limit of
     the indexed domain of second parameter

3  3.1 plot one curve by varying value of first parameter
   3.2 append to the curve nodes of current surface

4  Increment value of second parameter by trace magnitude
     of second parameter

5  If second parameter value >= upper limit, then
   5.1 set quit flag
   5.2 if parameter value > upper limit, then
         set to upper limit

6  If quit flag is raised, then
     goto step 7
   else
```

```
    goto step 3

7  Return current surface node
```

D.3.1.2 Unit-Length Tracing

Ullcurve() unit-length traces a curve on the parameter and the domain requested. Indices to the parameter and the domain it should trace are passed as arguments. Initial trial incremental step size is arbitrary. Currently it is set to one percent of domain interval.

Algorithm for ullcurve():

```
1  Create a new curve node, error exit if failure

2  Set current value of indexed parameter to lower limit
     of indexed domain

3  3.1 evaluate the variables
   3.2 create a point record
   3.3 append to the point nodes of current curve

4  Set trial incremental step size of indexed parameter to
     1% of the indexed domain interval

5  Call homein() to find the actual incremental step size
     which yields a point whose distance is (trace
     magnitude +/- trace precision) from current point,
     error exit if failure

6  If parameter value >= upper limit, then
   6.1 set quit flag
   6.2 if parameter value > upper limit, then
         set to upper limit

7  7.1 Evaluate the variables
   7.2 Create a point record
   7.3 Append to the point nodes of current curve

8  If quit flag is raised, then
   8.1 goto step 9
   else
   8.2.1 set trial incremental step size of indexed
           parameter to current actual step size
   8.2.2 goto step 5

9  Return current curve node
```

The philosophy of ullsurface() is similar to ullcurve().

Algorithm for ullsurface():

```
1  Create a new surface node, error exit if failure

2  Set current value of second parameter to lower limit of
     the indexed domain of second parameter

3  3.1 plot one curve by varying value of first parameter
   3.2 append to the curve nodes of current surface

4  Set trial incremental step size of second parameter to
     1% of the indexed domain interval

5  5.1 set first parameter to its lower limit
   5.2 call homein() to find the actual incremental step
         size which yields a point whose distance is
         (trace magnitude +/- trace precision) from
         current point, error exit if failure

6  6.1 increase second parameter value by actual step size
   6.2 if parameter value >= upper limit, then
       6.2.1 set quit flag
       6.2.2 if parameter value > upper limit, then
               set to upper limit

7  Call ullcurve to trace the curve where current value
     of second parameter is kept unchanged, error exit if
     failure

8  If quit flag is raised, then
   8.1 goto step 9
   else
   8.2.1 set trial incremental step size of second
           parameter to current actual step size
   8.2.2 goto step 5

9  Return current surface node
```

Both uslcurve and uslsurface use the routine homein() to find the actual step size of a parameter. Homein() takes the index values of a parameter and its current domain, and performs a binary expansion and contraction on the trial step size until the required point is located. The resulting point must lie within a distance of the trace magnitude from the current point. Since the parameter and the domain are indexed, homein() can perform the search on any of the domains of any of the parameters.

Algorithm for homein():

```
1  Set indexed parameter value to (base value + step size)
```

```
2  2.1 evaluate the variables
   2.2 calculate the distance from base point

3  If abs(distance - magnitude) <= precision then
     goto step 6

4  If (distance > magnitude then) then
     perform a binary contraction on step size
   else
     perform a binary expansion on step size

5  If looping counter reaches its limit then
     error exit
   else
     increase counter value and goto step 2

6  Return the value of current step size
```

Note that step 5 in the above algorithm is a control mechanism which limits the number of loops homein() can go through. If the step size is not found within the limit, then an error exit is forced.

D.3.1.3 Expression Evaluation

Evaluation of an expression is also recursive. The current node is considered first. If it is a leaf node, then the value of the node is returned. Otherwise the values of the left and right subtrees are evaluated. Then the result of the operation is returned. For example, it current node is a "addop" node, then the sum or difference of the values of the two subtrees are returned. Evalu() is the lead module to expression evaluation and evalu2() is the recursive routine which evaluate expression subtrees.

D.4 Output section

D.4.1 EXPORTING TO UNIGRAFIX FORMAT [PHASE 5]

After the tracing section is completed, the leading curve node or the leading surface node is returned to main(). Main() passes this node to the leading module of the output section, export(). Export() translates the hierarchy structure into UNIGRAFIX descriptive format and puts the translated script into files. Two files are used. Points (UNIGRAFIX vertices) are written to one. Curves and surfaces (UNIGRAFIX wires and faces) are written to another. After the translation is completed, the second file is appended to the first, giving one output file ready for display.

```
Algorithm for export():

1  Open two files, error exit if failure

2  Initialize all counters

3  Write to file 1, the min and max of each variable

4  Translate the hierarchy structure:
   if it is a curve
   4.1 while there is a curve node do
         call putcurve()
   else if it is a surface
   4.2 while there is a surface node do
         call putsurface()

5  If axis mode is on, define the axes in terms of the min
     and max of the variables, and write to file 1

6  Append file 2 to file 1
```

The counters are used to assign unique names for vertices, wires and faces. The counters are hexadecimal. Vertex names are prefixed with 'V', wire names are prefixed with 'W', and face names are prefixed with 'F'.

Putcurve() is given a sequence of curve nodes. The sequence of points in each curve node is written out in UNIGRAFIX format. The coordinates of the points are written to file 1, while the sequencing of the points which forms a curve is written to file 2.

```
Algorithm for putcurve():

1  If there are no more points, then goto step 4

2  Write a VERTEX statement to file 1 defining the
     coordinates of current point

3  If the WIRE statement is not too long, then
   3.1.1 append the current vertex name to the current
           wire statment
   3.1.2 increase number of segments in current wire by 1
   else
   3.2.1 close current WIRE statement
   3.2.2 open a new WIRE statement
   3.2.3 append the current vertex name to the current
           wire
   3.1.4 reset number of segments in current wire to 0

4  4.1 if number of segments in current wire is 0, then
         append again current vertex name to wire
```

```
    4.2 close current WIRE statement

5   Return
```

The maximum number of segments a wire contains is arbitrary. Any number between 5 to 20 is sound and reasonable.

A surface consists of a sequence of curve nodes. Two successive curve nodes, referred to as low and high, are considered each time. The sequences of points on the two curves form either meshes or patches that make up the surface.

Putsurface() is given a sequence of surface nodes. The coordinates of the points are written to file 1, while the sequencing of the points which forms the meshes or the patches are written to file 2.

Algorithm for putsurface():

```
1   If there are no more curves, then
      goto step 9
    else
      set low curve to current curve

2   Dump points in low curve to file 1

3   Set high curve to low curve

4   If there is no curve following high curve, then
      goto step 9
    else
      set high curve to curve following high curve

5   Dump points in high curve to file 1

6   If mesh display mode, then
      display low curve as UNIGRAFIX wires, and dump to
        file 2
    else
      display low curve and high curve as UNIGRAFIX faces,
        and dump to file 2

7   7.1 set low curve to current high curve
    7.2 goto step 4

8   If mesh display mode, then
      display last curve as UNIGRAFIX wires, and dump to
        file 2

9   Return
```

D.5 Function directory

audit()	-- dump to logfile values of identifier records
chkcmdln()	-- report the status of various switches
chkspec()	-- apply overall checking to specification
evalu()	-- lead module to expression evaluation
evalu2()	-- evaluate expression subtrees
export()	-- lead module of output section
findulen()	-- find default tracing magnitude for each domain
getarg()	-- take a line & split it into an array of words
getcmdln()	-- take an array of string & scan its contents
getconst()	-- get constant declaration & definition
getdom()	-- get domain declaration & definition
geteqt()	-- get equation definition
getpar()	-- get parameter declaration
getspec()	-- read specification from input stream
getoken()	-- get a constant or a number
getvar()	-- get variable declaration
homein()	-- find the actual step size in unit-length tracing
intrspec()	-- interpret shell for specification parsing
isaddop()	-- check for an addition operator
isargu()	-- check for an argument
isconstant()	-- check if the given word is "constant"
isdomain()	-- check if the given word is "domain"
isexpr()	-- check for an expression
isfact1()	-- check for a level one factor
isfact2()	-- check for a level two factor
isform()	-- lead module to expression parsing
isfunc()	-- check for a function
isiden()	-- check for an identifier
isletter()	-- check if given 2 characters are identical
islit()	-- check for a literal
ismultop()	-- check for a multiplication operator
isparameter()	-- check if the given word is "parameter"
ispowop()	-- check for a power operator
isterm()	-- check for a term
isvariable()	-- check if the given word is "variable"
logcurve()	-- dump to logfile the contents of a curve node
logpoint()	-- dump to logfile the contents of a point node
logsurface()	-- dump to logfile the contents of a surface node
main()	-- main module
message()	-- lead module to error exit
mkcurve	-- make a curve record
mkpoint	-- make a point record
mksurface	-- make a surface record
namegood()	-- verify that the syntax of an identifier
nameused()	-- verify that an identifier is declared or not
nextid()	-- advance the given counter
plotcurves()	-- curve plot shell
plotspec()	-- lead module of plot shell

```
plotsurfaces() -- surface plot shell
prtexpr()      -- dump to logfile the layout of an expression tree
prtop()        -- dump to logfile the contents of a tree node
putaxis()      -- include axes to the UNIGRAFIX output file
putcmap()      -- include an UNIGRAFIX colour definition
putcurve()     -- put curve into UNIGRAFIX format
putlimit()     -- put limits of the variables in UNIGRAFIX format
putsurface()   -- put surface into UNIGRAFIX format
ullcurve()     -- unit length trace 1 domain interval
ullsurface()   -- unit length trace 1 domain combination
uslcurve()     -- unit step trace 1 domain interval
uslsurface()   -- unit step trace 1 domain combination
```

Appendix E

SOURCE CODE

FOR UGTRACE

```
/****************************************************************************
 *                                                                          *
 *  cmdln.h -- header file for command line parsing                         *
 *                                                                          *
 *  Author: Henri Cheung at McMaster University, Hamilton, Ontario          *
 *                                                                          *
 ****************************************************************************/


                       /*   optflags value    */

#define  OPTON   0x001 /* 0000 0000 0000 0001  option encountered        */
#define  ARGOK   0x002 /* 0000 0000 0000 0010  argument received         */
#define  OPTON2  0x004 /* 0000 0000 0000 0100  more than once            */

#define  SYNERR  0x100 /* 0000 0001 0000 0000  unresolved syntax         */
```

```
/*****************************************************************************
 *                                                                          *
 *  evalu.h -- header file for evaluating of expression                     *
 *                                                                          *
 *  Author: Henri Cheung at McMaster University, Hamilton, Ontario          *
 *                                                                          *
 *****************************************************************************/


                        /* flags for tolerable errors */

#define ENEGP   0x004   /* 0000 0000 0000 0100   -ve # raised to real power */

                        /* flags for intolerable errors: program must stop  */

#define ENUL    0x100   /* 0000 0001 0000 0000   null pointer encountered   */
#define EZERO   0x200   /* 0000 0010 0000 0000   zero divisor               */


#define SPLIM  1.0E10   /* [system dependent] max representable real number */
#define SNLIM 1.0E-10   /* [system dependent] min representable real number */
```

```
/*****************************************************************************
 *                                                                           *
 *  export.h -- header file for generating of UNIGRAFIX format file          *
 *                                                                           *
 *  Author: Henri Cheung at McMaster University, Hamilton, Ontario           *
 *                                                                           *
 *****************************************************************************/


#define  IDZERO  "0000000000000000000000000000000000000000000000000000"

#define  IDWIDTH   8
#define  WSEGLEN   8
```

```
/*****************************************************************************
 *                                                                           *
 *  global.h -- header file for global definitions                           *
 *                                                                           *
 *  Author: Henri Cheung at McMaster University, Hamilton, Ontario           *
 *                                                                           *
 *****************************************************************************/


#define  LATTICE     0   /* Lattice C compiler on PC                          */
#define  UNIX        1   /* CC compiler on UNIX                               */
#define  VER      UNIX   /* compiler flag                                     */

#define  VERSION   1.0   /* version number                                    */

#define  WORDSIZ    32   /* max. word size defined for parsing                */
#define  WORDBUFF  128   /* array dimension defined for parsing               */

#define  STRSIZ    256   /* max. string length defined for parsing            */
#define  STRBUFF   128   /* array dimension defined for parsing               */

#define  NUMCONST   64   /* max # of constants allowed [arbitrary]            */
#define  NUMPARAM    8   /* max # of parameters allowed                       */
#define  NUMVAR      8   /* max # of variables allowed                        */
#define  NUMDOM     32   /* max # of partitions allowed for each parameters   */

#define  OFF         0
#define  ON          1

#define  NO          0
#define  YES         1

#define  FALSE       0
#define  TRUE        1
```

```
/******************************************************************************
 *                                                                            *
 *  message.h -- header file for message                                      *
 *                                                                            *
 *  Author: Henri Cheung at McMaster University, Hamilton, Ontario            *
 *                                                                            *
 ******************************************************************************/


#define MSGXXX "\n...aborted. \n"
#define MSGYYY "\n\t :unknown message number. \n"

        /* Phase 1 Messages */

#define MSG001 "\n\t :source filename missing. \n"
#define MSG002 "\n\t :output filename missing. \n"
#define MSG003 "\n\t :unresolved argument(s) ignored. \n"
#define MSG004 "\n\t :fi option encountered again, previous one forgotten. \n"
#define MSG005 "\n\t :fo option encountered again, previous one forgotten. \n"
#define MSG006 "\n\t -logging is on. \n"
#define MSG007 "\n\t -unit length tracing. \n"
#define MSG009 "\n\t -mesh display. \n"
#define MSG010 "\n\t -axis are plotted. \n"

        /* Phase 2 and 3 Messages */

#define MSG008 "\n\t :word too long, truncated to WORDSIZ. \n"
#define MSG500 "\n\t :cannot open temporary work file. \n"
#define MSG501 "\n\t :too many lines. \n"
#define MSG502 "\n\t :cannot open source file. \n"
#define MSG503 "\n\t :cannot open output file. \n"
#define MSG504 "\n\t :insufficient memory. \n"
#define MSG505 "\n\t :fail to release memory block. \n"
#define MSG506 "\n\t :too few lines. \n"
#define MSG507 "\n\t :matching '{' not found. \n"
#define MSG508 "\n\t :matching '}' not found. \n"
#define MSG509 "\n\t :too many words. \n"
#define MSG510 "\n\t :cannot open log file. \n"

#define MSG511 "\n\t :constant definition syntax. \n"
#define MSG512 "\n\t :bad constant name. \n"
#define MSG513 "\n\t :invalid expression(s). \n"
#define MSG514 "\n\t :too many constant(s) defined. \n"
#define MSG515 "\n\t :declaring constant already declared as variable. \n"
#define MSG516 "\n\t :declaring constant already declared as parameter. \n"
#define MSG517 "\n\t :declaring constant already previously declared. \n"

#define MSG521 "\n\t :variable declaration syntax. \n"
#define MSG522 "\n\t :invalid identifier(s). \n"
#define MSG523 "\n\t :too many variable(s). \n"
#define MSG524 "\n\t :declaring variable already declared as constant. \n"
#define MSG525 "\n\t :declaring variable already declared as parameter. \n"
#define MSG526 "\n\t :declaring variable already previously declared. \n"
#define MSG527 "\n\t :variable already previously defined. \n"

#define MSG531 "\n\t :parameter declaration syntax. \n"
```

```
#define MSG532 "\n\t :invalid identifier(s). \n"
#define MSG533 "\n\t :too many parameter(s). \n"
#define MSG534 "\n\t :declaring parameter already declared as constant. \n"
#define MSG535 "\n\t :declaring parameter already declared as variable. \n"
#define MSG536 "\n\t :declaring parameter already previously declared. \n"

#define MSG541 "\n\t :domain specification syntax. \n"
#define MSG542 "\n\t :invalid domain range. \n"
#define MSG543 "\n\t :undefined parameter(s). \n"
#define MSG544 "\n\t :too many domain(s). \n"
#define MSG545 "\n\t :point domain(s). \n"

#define MSG551 "\n\t :invalid formula(s). \n"
#define MSG552 "\n\t :invalid expression(s). \n"
#define MSG553 "\n\t :invalid term(s). \n"
#define MSG554 "\n\t :invalid factor(s). \n"
#define MSG555 "\n\t :invalid argument(s). \n"
#define MSG556 "\n\t :invalid identifier(s). \n"
#define MSG557 "\n\t :missing ')'. \n"

#define MSG561 "\n\t :tracing specification syntax.\n"
#define MSG571 "\n\t :display specification syntax.\n"

#define MSG581 "\n\t :equation syntax. \n"
#define MSG582 "\n\t :unresolved symbol(s). \n"
#define MSG583 "\n\t :invalid expression(s). \n"

#define MSG591 "\n\t :precision specification syntax. \n"
#define MSG592 "\n\t :sensitivity syntax. \n"

#define MSG101 "\n\t :absolute of trace speed taken. \n"
#define MSG102 "\n\t :absolute of display interval taken. \n"
#define MSG103 "\n\t :absolute of precision taken. \n"
#define MSG104 "\n\t :absolute of sensitivity taken. \n"
#define MSG105 "\n\t -trace interval set to default value. \n"
#define MSG106 "\n\t -display interval set to trace speed. \n"
#define MSG107 "\n\t -precision set to default: 1%% of trace. \n"
#define MSG108 "\n\t -sensitivity set to default: 0.1%% of domain range. \n"
#define MSG601 "\n\t :no parameter declared. \n"
#define MSG602 "\n\t :param(s) more than var(s). \n"
#define MSG603 "\n\t :missing parameter domain(s). \n"
#define MSG604 "\n\t :missing equation(s). \n"
#define MSG605 "\n\t :no variable declared. \n"
#define MSG606 "\n\t :too many parameter declared. \n"
#define MSG607 "\n\t :too many variable declared. \n"

        /* Phase 4 and 5 Messages */

#define MSG201 "\n\t :incomplete trace - cannot get closer to asymptotes. \n"

#define MSG251 "\n\t :too many loops, probably oscillating. \n"
#define MSG252 "\n\t :exit conditions not met even increment is zero. \n"

#define MSG701 "\n\t :unable to calculate first point. \n"
#define MSG702 "\n\t :expand IDWIDTH and try again. \n"
#define MSG703 "\n\t :error in calculating expression. \n"
```

```
#define MSG751 "\n\t :current curve is partial. \n"
#define MSG752 "\n\t :current surface is partial. \n"
```

```
/**************************************************************************
 *                                                                        *
 *  struct.h -- header file for misc. structure definitions               *
 *                                                                        *
 *  Author: Henri Cheung at McMaster University, Hamilton, Ontario        *
 *                                                                        *
 **************************************************************************/


/**************************************************************************
 *                                                                        *
 *  information node for an expression tree                               *
 *                                                                        *
 **************************************************************************/


union cell
{
  int    ival;              /* integer value                   */
  double rval;              /* real value                      */
  struct identifier *ptr;   /* pointer to constant/parameter */
  char   sval[WORDSIZ];     /* symbol value                    */
};


/**************************************************************************
 *                                                                        *
 *  tree structure for an expression & related flags                       *
 *                                                                        *
 **************************************************************************/


#define  INTEGER   0x01       /* 0000 0000 0000 0001  */
#define  REAL      0x02       /* 0000 0000 0000 0010  */
#define  OPERATOR  0x08       /* 0000 0000 0000 1000  */
#define  FUNCTION  0x10       /* 0000 0000 0001 0000  */
#define  CONSTID   0x20       /* 0000 0000 0010 0000  */
#define  PARAMID   0x40       /* 0000 0000 0100 0000  */
#define  VARID     0x80       /* 0000 0000 1000 0000  */

struct expr
{
  int          ctype  ;       /* cell type indicator  */
  union cell   info   ;       /* cell content         */
  struct expr  *l, *r ;       /* left & right pointer */
};

#define  EXPRSIZ   sizeof(struct expr)


/**************************************************************************
 *                                                                        *
 *  record for an identifier definition & related flags                    *
 *                                                                        *
 **************************************************************************/
```

```
#define  CONSTANT     0x01   /* 0000 0000 0000 0001  declared    */
#define  PARAMETER    0x02   /* 0000 0000 0000 0010  declared    */
#define  VARIABLE     0x04   /* 0000 0000 0000 0100  declared    */

#define  USEDFLAG     0x10   /* 0000 0000 0001 0000  used        */
#define  DOMFLAG      0x20   /* 0000 0000 0010 0000  domained    */


struct identifier
{
  int         itype         ;  /* identifier type              */
  char        nam[WORDSIZ]  ;  /* identifier name              */
  double      valu          ;  /* current value                */
  double      vmin, vmax    ;  /* min value, max value         */
  struct expr *f            ;  /* equivalent expression, if any */
};

#define  IDENSIZ   sizeof(struct identifier)


/*****************************************************************************
 *                                                                           *
 *  record for a parameter partition & related flags                         *
 *                                                                           *
 *****************************************************************************/


#define  LESS_THAN     0x1   /* 0000 0000 0000 0001  (<)   */
#define  GREATER_THAN  0x2   /* 0000 0000 0000 0010  (>)   */
#define  EQUAL_TO      0x4   /* 0000 0000 0000 0100  (=)   */

struct domain
{
  int    flag        ;  /*                       */
  double dmin, dmax  ;  /* min value, max value */
  double trace       ;  /* trace interval       */
  double display     ;  /* display interval     */
  double precision   ;  /* precision            */
  double sensitivity ;  /* sensitivity          */
};

typedef struct domain domarr[NUMDOM];  /* array of domain record */

#define  DOMSIZ   sizeof(struct domain)
```

```
/*************************************************************************
 *                                                                       *
 *  struct2.h -- header file for structure definitions used in plotting  *
 *                                                                       *
 *  Author: Henri Cheung at McMaster University, Hamilton, Ontario       *
 *                                                                       *
 *************************************************************************/


#define  LOOPMAX 127
#define  ERRLOOP   1
#define  ERRINCR   2
#define  ERRMEM    3


/*************************************************************************
 *                                                                       *
 *  record for a point in space                                          *
 *                                                                       *
 *************************************************************************/


#define PARMAX 2
#define VARMAX 3


struct point
{
  int     pflag ;
  double  u[PARMAX] ;
  double  x[VARMAX] ;
  struct point *np  ;
};

#define  PTSIZ  sizeof(struct point)


/*************************************************************************
 *                                                                       *
 *  record for a curve in space                                          *
 *                                                                       *
 *************************************************************************/


struct curve
{
  int    cflag ;
  int    i1,i2 ;  /* i1-th parameter, i2-th domain */
  double nump  ;
  struct point *head ;
  struct curve *nc   ;
};

#define  CURVESIZ  sizeof(struct curve)
```

```
/**************************************************************************
 *                                                                        *
 *  record for a surface in space                                         *
 *                                                                        *
 **************************************************************************/


struct surface
{
  int    sflag ;
  int    i1,i2 ;  /* i1-th domain of 0th parameter, i2-th of 1st */
  double nump, numc    ;
  struct curve   *root ;
  struct surface *ns   ;
};

#define  SURFASIZ  sizeof(struct surface)
```

```
/***************************************************************************
 *                                                                         *
 *  ugtrace.c -- driver of UGTRACE                                         *
 *                                                                         *
 *  The program consists of 5 phases:                                      *
 *                                                                         *
 *  [Phase 1] Getting command line & setting i/o streams                   *
 *  [Phase 2] Reading plot specifications                                  *
 *  [Phase 3] Parsing plot specifications                                  *
 *  [Phase 4] Generating points                                            *
 *  [Phase 5] Converting points to UNIGRAFIX format [curves & surfaces]    *
 *                                                                         *
 *  Author: Henri Cheung at McMaster University, Hamilton, Ontario         *
 *                                                                         *
 ***************************************************************************/


#include  <stdio.h>
#include  "global.h"

extern int  nparam, nvar ;
extern char infile[], outfile[] ;

extern int optlog;
extern FILE *logf;

char *logID;


main (argc,argv)
int  argc    ;
char *argv[] ;
{ int    nl = 0 ;
  char *mktemp(), *malloc() ;
  double np = 0.0, plotspec() ;

  /* PHASE 0 : SETTING THINGS UP                */

  if (argc == 1)
  {
    fprintf(stderr,"\nUsage: ugtrace [-a] [-us|-ul] [-dm|-dp] [-fi file] [-fo file]\n"
    goto ugx;
  }

  fprintf(stderr,"\nUGTRACE Version %3.1f \n",VERSION);
  fprintf(stderr,"\nProgram Running...    \n"        );

  if ((logID = malloc(STRSIZ)) == NULL)  message(504);
#if VER == LATTICE
  strcpy(logID,"logfile");
#endif
#if VER == UNIX
  strcpy(logID,"logXXXXXX");
  logID = mktemp(logID);
#endif
```

```
/* PHASE 1 : SCANNING COMMAND LINE */

fprintf(stderr,"\nPhase 1 : Scanning Command Line \n");

getcmdln(argc,argv);

if (optlog == ON)
{
  if ((logf = fopen(logID,"w")) == NULL)  message(510);
  fprintf(logf,"\nUGTRACE Version %3.1f \n",VERSION);
  fprintf(logf,"\nProgram Running...    \n"        );
  fprintf(logf,"\nPhase 1 : Scanning Command Line \n");  fflush(logf);
}

chkcmdln();


/* PHASE 2 : LOADING SPECIFICATION */

fprintf(stderr,"\nPhase 2 : Loading Specification [from %s] \n",infile);

if (optlog == ON)
{
  fprintf(logf,"\nPhase 2 : Loading Specification [from %s] \n",infile);
  fflush(logf);
}

nl = getspec(infile);


/* PHASE 3 : PARSING SPECIFICATION */

fprintf(stderr,"\nPhase 3 : Parsing Specification [%d line(s)] \n",nl);

if (optlog == ON)
{
  fprintf(logf,"\nPhase 3 : Parsing Specification [%d line(s)] \n",nl);
  fflush(logf);
}

intrspec(nl);
chkspec();
audit();


/* PHASE 4 : TRACING */

fprintf(stderr,"\nPhase 4 : ");
fprintf(stderr,"Tracing [%d degree, %d dimension] \n",nparam,nvar);

if (optlog == ON)
{
  fprintf(logf,"\nPhase 4 : ");
  fprintf(logf,"Tracing [%d degree, %d dimension] \n",nparam,nvar);
  fflush(logf);
  }

  np = plotspec();

  fprintf(stderr,"\n  --> %10.1f point(s) in memory \n",np);

  if (optlog == ON)
  {
    fprintf(logf,"\n  --> %10.1f point(s) in memory \n",np);
  }


  /* PHASE 5 : EXPORTING TO UNIGRAFIX FORMAT */

  fprintf(stderr,"\nPhase 5 : Exporting in UNIGRAFIX format [to %s] \n",outfile);

  if (optlog == ON)
  {
    fprintf(logf,"\nPhase 5 : Exporting in UNIGRAFIX format [to %s] \n",outfile);
    fflush(logf);
  }

  export(outfile);


  /* PHASE 6 : CLOSING PROCEDURES */

  fprintf(stderr,"\n...Completed. \n\n");

  if (optlog == ON)
  {
    fprintf(stderr,"[Logfile is %s]\n\n",logID);
    fprintf(logf,"\n...Completed. \n\n[Logfile is %s].\n\n",logID);
    fclose(logf);
  }

ugx:
  ;
}
```

```
/*****************************************************************************
 *                                                                           *
 *  cmdln.c -- misc functions for command line parsing                       *
 *                                                                           *
 *  Author: Henri Cheung at McMaster University, Hamilton, Ontario           *
 *                                                                           *
 *****************************************************************************/


#include  <stdio.h>
#include  "global.h"
#include  "cmdln.h"


char infile[STRSIZ]  ;    /* character string for input source name  */
char outfile[STRSIZ] ;    /* character string for output target name */
int  optfi   = OFF ;
int  optfo   = OFF ;
int  optulen = OFF ;
int  optmesh = OFF ;
int  optaxis = OFF ;
int  optsyn  = OFF ;
int  optlog  = OFF ;
FILE *logf   ;


/*****************************************************************************
 *                                                                           *
 *  getcmdln -- parse command line argument, set input & output stream       *
 *                                                                           *
 *****************************************************************************/

getcmdln (argc,argv)
int   argc   ;
char *argv[] ;
{ int  i = 0 ;
  FILE *fi, *fo;

gl1:

  /* treat command line parameters */

  if (++i >= argc)  goto gl2 ;

  if (strcmp(argv[i],"-fi") == 0)
  {
    strcpy(infile,"stdin" );
    optfi |= ((optfi & OPTON) != OFF) ? OPTON2 : 0 ;
    optfi |= OPTON ;

    if (i+1 >= argc)                optfi &= ~ARGOK;
    else if (argv[i+1][0] == '-')   optfi &= ~ARGOK;
    else
    {
      strcpy(infile,argv[++i]);     optfi |=  ARGOK;
    }
  goto gll;
}

if (strcmp(argv[i],"-fo") == 0)
{
  strcpy(outfile,"stdout");
  optfo |= ((optfo & OPTON) != OFF) ? OPTON2 : 0 ;
  optfo |= OPTON ;

  if (i+1 >= argc)                  optfo &= ~ARGOK;
  else if (argv[i+1][0] == '-')     optfo &= ~ARGOK;
  else
  {
    strcpy(outfile,argv[++i]) ;     optfo |=  ARGOK;
  }
  goto gll;
}

if (strcmp(argv[i],"-nl") == 0)
{
  optlog = OFF ;    goto gll;
}

if (strcmp(argv[i],"-l") == 0)
{
  optlog = ON ;     goto gll;
}

if (strcmp(argv[i],"-na") == 0)
{
  optaxis = OFF;    goto gll;
}

if (strcmp(argv[i],"-a") == 0)
{
  optaxis = ON ;    goto gll;
}

if (strcmp(argv[i],"-us") == 0)
{
  optulen = OFF;    goto gll;
}

if (strcmp(argv[i],"-ul") == 0)
{
  optulen = ON ;    goto gll;
}

if (strcmp(argv[i],"-dp") == 0)
{
  optmesh = OFF;    goto gll;
}

if (strcmp(argv[i],"-dm") == 0)
{
  optmesh = ON ;    goto gll;
  }

  optsyn |= SYNERR;  /* syntax error */

  goto g11 ;

g12:
  ;

}


/**************************************************************************
 *                                                                        *
 *  chkcmdln -- check command line argument                               *
 *                                                                        *
 **************************************************************************/

chkcmdln ()
{ FILE *fi, *fo;

  if (optulen == ON)    message(7);
  if (optmesh == ON)    message(9);
  if (optaxis == ON)    message(10);
  if (optlog  == ON)    message(6);

  /* error treatment */

  if ((optsyn & SYNERR) != OFF) message(3);

  if ((optfi & OPTON2) != OFF)                              message(4);
  if(((optfi & OPTON)  != OFF) && ((optfi & ARGOK) == OFF)) message(1);

  if ((optfo & OPTON2) != OFF)                              message(5);
  if(((optfo & OPTON)  != OFF) && ((optfo & ARGOK) == OFF)) message(2);

  if ((optfi & ARGOK) == OFF) strcpy(infile, "stdin" );
  if ((optfo & ARGOK) == OFF) strcpy(outfile,"stdout");

  /* try to access of i/o files */

  if (strcmp(infile,"stdin") != 0)
  if ((fi = fopen(infile,"r")) == NULL) message(502);
  else fclose(fi);

  if (strcmp(outfile,"stdout") != 0)
  if ((fo = fopen(outfile,"a+")) == NULL) message(503);
  else fclose(fo);

}
```

```
/****************************************************************************
 *                                                                          *
 *  input.c -- misc functions for input parsing                             *
 *                                                                          *
 *  Author: Henri Cheung at McMaster University, Hamilton, Ontario          *
 *                                                                          *
 ****************************************************************************/


#include  <stdio.h>
#include  <ctype.h>
#include  "global.h"

extern FILE *logf ;
extern int  optlog;

char *buffer   ;  /* buffer is an array of lines    */
char *argument ;  /* argument is an array of words */


/****************************************************************************
 *                                                                          *
 *  getspec -- read from input stream                                       *
 *                                                                          *
 ****************************************************************************/

getspec (fn)
char fn[];
{ int  j, k, nc, nl = 0, flag = OFF ;
  char *l, *c, *calloc() ;
  FILE *fin ;

  /* allocate transitional memory buffer */

  if ((buffer = calloc(STRBUFF,STRSIZ)) == NULL)  message(504);

  /* prepare source stream, proceed no further if cannot open */
  /* obtain whole file into memory buffer: STRBUFF x STRSIZ   */

  fin = (strcmp(fn,"stdin") == 0) ? stdin : fopen(fn,"r") ;
  if (fin == NULL)  message(502) ;

  l = buffer ;
  while ((nl < STRBUFF) && (fgets(l,STRSIZ,fin) != NULL))
  {
    l += STRSIZ ;  ++nl ;
  }
  fclose(fin) ;

  if (nl >= STRBUFF)  message(501) ;
  if (nl <= 0)        message(506) ;

  /* replace comment block by blanks */

  for (j = 0; j < nl; j++)
  {
    nc = strlen(buffer + j * STRSIZ) ;
    for (k = 0; k < nc; k++)
    switch ( *(c = buffer+j*STRSIZ+k) )
    {
      case '{' :                      flag += ON;  *c = ' ';    break;
      case '}' : if (flag != OFF)   { flag -= ON;  *c = ' '; }
                 else               message(507);               break;
      case '\t':                                   *c = ' ';    break;
      case '\n':                                                break;
      default  : if (flag != OFF)                  *c = ' ';    break;
    }
  }
  if (flag != OFF)  message(508);

  return(nl);
}


/*******************************************************************************
 *                                                                             *
 *  intrspec -- parse input                                                    *
 *                                                                             *
 *******************************************************************************/

intrspec (nl)
int nl;
{ char *calloc() ;
  int  j, k, nw, flag = OFF ;

  if (nl == 0)  message(506);

  /* allocate transitional memory buffer */

  if ((argument = calloc(WORDBUFF,WORDSIZ)) == NULL)  message(504);

  /* break down into arguments & parse */

  for (j = 0; j < nl; j++)
  if ((nw = getarg(argument,buffer+j*STRSIZ)) > 0)
  {
    if (optlog == ON)
    {
      for (k = 0;  k < nw;  k++)  fprintf(logf,"|%s",argument+k*WORDSIZ);
      fprintf(logf,"[%d]\n",nw);  fflush(logf);
    }

    if (isconstant(argument)  != FALSE) { getconst(nw,argument); goto spl; }
    if (isparameter(argument) != FALSE) { getpar(nw,argument);   goto spl; }
    if (isvariable(argument)  != FALSE) { getvar(nw,argument);   goto spl; }
    if (isdomain(argument)    != FALSE) { getdom(nw,argument);   goto spl; }
    geteqt(nw,argument);
spl:
    ;
  }

  /* release transitional memory block */
```

```
#if VER == LATTICE  /*  Lattice C version */
  if (free(buffer  ) !=   0)  message(505);
  if (free(argument) !=   0)  message(505);
#endif
#if VER == UNIX     /*  Unix version      */
  if (free(buffer  ) != 239)  message(505);
  if (free(argument) != 239)  message(505);
#endif
}


/*****************************************************************************
 *                                                                           *
 *  getarg -- split string [%s\0] into an array of words,                   *
 *             a null string is appended as the last array element           *
 *                                                                           *
 *****************************************************************************/

getarg (argvec,s)
char *argvec, *s;
{ int  inword = NO, k = 0, nw = 0;

ga1:

  if (nw >= WORDBUFF)  message(509);

  if (*s == '\0')  goto ga2;

  switch (*s)
  {
    case ' ':  case '\t':  case '\n':  /* these characters are skipped */

      if (inword == YES)
      {
        *(argvec+k) = '\0';     k = 0 ;
        argvec += WORDSIZ ;     nw++  ;
        inword = NO ;
      }
      break;

    case '+': case '-': case '*': case '/': case '^':
    case '(': case ')':
    case '=': case ',':     /* these characters forced a separate word */

      if (inword == YES)
      {
        *(argvec+k) = '\0';     k = 0 ;
        argvec += WORDSIZ ;     nw++  ;
        inword = NO;
      }

      *argvec = *s;             k++   ;
      *(argvec+k) = '\0';       k = 0 ;
      argvec += WORDSIZ ;       nw++  ;
      break;
  case '<':  case '>':    /* these characters forced a separate word */

    if (inword == YES)
    {
      *(argvec+k) = '\0';     k = 0 ;
      argvec += WORDSIZ ;     nw++  ;
      inword = NO;
    }

    *argvec = *s;             k++   ;
    if (*(s+1) == '=')
    {
      s++;
      *(argvec+k) = *s ;      k++   ;
    }
    *(argvec+k) = '\0';       k = 0 ;
    argvec += WORDSIZ ;       nw++  ;
    break;

  case '.': case '_':     /* acceptable non-alphanumerics for word */
    if (k < WORDSIZ)
    {
      *(argvec+k) = *s ;      k++   ;
      inword = YES;
    }
    else message(8);
    break;

  default:

    if (isalnum(*s) == FALSE)  /* force a new word */
    {
      if (inword == YES)
      {
        *(argvec+k) = '\0';     k = 0 ;
        argvec += WORDSIZ ;     nw++  ;
        inword = NO;
      }

      *argvec = *s;             k++   ;
      *(argvec+k) = '\0';       k = 0 ;
      argvec += WORDSIZ ;       nw++  ;
    }
    else if (k < WORDSIZ)
    {
      *(argvec+k) = *s ;        k++   ;
      inword = YES;
    }
    else message(8);
    break;
}

s++;
goto gal;
```

```
ga2:

  if (inword == YES)
  {
    *(argvec+k) = '\0';
    argvec += WORDSIZ ;     nw++ ;
  }

  if (nw >= WORDBUFF)  message(509);
  *argvec = '\0';

  return(nw);
}
```

```
/******************************************************************************
 *                                                                            *
 *  parse.c -- misc functions for specification parsing                      *
 *                                                                            *
 *  Author: Henri Cheung at McMaster University, Hamilton, Ontario            *
 *                                                                            *
 ******************************************************************************/


#include  <stdio.h>
#include  <ctype.h>
#include  <math.h>
#include  "global.h"
#include  "struct.h"

extern FILE *logf ;
extern int   optlog, optulen;

int nconst = 0;                            /* total # of const defined & used */
int nparam = 0;                            /* total # of par   defined & used */
int nvar   = 0;                            /* total # of var   defined & used */

struct identifier *const[NUMCONST];        /* array for constant definitions  */
struct identifier *param[NUMPARAM];        /* array for parameter definitions */
struct identifier *var[NUMVAR]    ;        /* array for variable definitions  */

struct domain (*dom[NUMPARAM])[NUMDOM];    /* array for domain partitions     */


/******************************************************************************
 *                                                                            *
 *  getconst -- get constant declaration & definition                         *
 *                                                                            *
 ******************************************************************************/

getconst (n,bufp)
int   n ;
char *bufp ;
{ struct expr *f, *isform(), *reclaim() ;
  char *w, *malloc() ;
  double evalu() ;
  int j, flags ;

  if (n < 4)  message(511);  /* check # of arguments */

  /* is identifier syntax valid ? */

  w = (bufp += WORDSIZ);
  if (namegood(w) != TRUE)  message(512);

  /* is the name already declared ? */

  switch(nameused(w))
  {
    case CONSTANT :  message(517);  break;
    case PARAMETER:  message(516);  break;
    case VARIABLE :  message(515);  break;
    default: /* name not declared */break;
   }

   if (nconst >= NUMCONST)  message(514);

   /* check existence of '=' */

   if (*(bufp+=WORDSIZ) != '=')  message(511);

   /* check if specified expression is valid */

   if ((f = isform(bufp+=WORDSIZ)) == NULL)  message(513);

   /* allocate new slot in the "const" array */

   j = nconst ;  nconst++ ;

   if ((const[j] = (struct identifier *) malloc(IDENSIZ)) == NULL)
   message(504);

   strcpy(const[j]->nam,w) ;
   const[j]->itype = CONSTANT ;
   const[j]->f     = f        ;
   const[j]->valu  = evalu(f) ;
   const[j]->vmin  = const[j]->vmax = 0.0 ;

   if (optlog == ON)
   {
     fprintf(logf,"[%d]%s = %f\n",j,const[j]->nam,const[j]->valu);
     fflush(logf);
   }

 }


/*****************************************************************************
 *                                                                           *
 *  getvar -- get variable declaration                                       *
 *                                                                           *
 *****************************************************************************/

getvar (n,bufp)
int   n ;
char *bufp ;
{ int j, flags = OFF ;
  char *w, *malloc() ;
  struct expr *reclaim() ;

  if (n < 2)  message(521);  /* check # of arguments */

gvl:

  /* is identifier syntax valid ? */

  w = (bufp += WORDSIZ);
  if (namegood(w) != TRUE)  message(522);

  /* is the name already declared ? */

  switch(nameused(w))
  {
    case CONSTANT :  message(524);  break;
    case PARAMETER:  message(525);  break;
    case VARIABLE :  message(526);  break;
    default: /* name not declared */break;
  }

  /* allocate new slot in the "var" array */

  if (nvar >= NUMVAR)  message(523);

  j = nvar ;    nvar++ ;

  if ((var[j] = (struct identifier *) malloc(IDENSIZ)) == NULL)
  message(504);

  strcpy(var[j]->nam,w) ;
  var[j]->itype = VARIABLE          ;
  var[j]->f     = NULL              ;
  var[j]->valu  = 0.0               ;
  var[j]->vmin  = var[j]->vmax = 0.0 ;

  if (optlog == ON)
  {
    fprintf(logf,"[%d]%s = %f\n",j,var[j]->nam,var[j]->valu); fflush(logf);
  }

  bufp += WORDSIZ ;
  if (strcmp(bufp,"")  != 0)
  if (strcmp(bufp,",") != 0)  message(521) ;
  else                        goto gvl     ;
}


/*******************************************************************************
 *                                                                             *
 *  getpar -- get parameter declaration                                        *
 *                                                                             *
 *******************************************************************************/

getpar (n,bufp)
int   n ;
char *bufp ;
{ int j, flags = OFF ;
  struct expr *reclaim() ;
  char *w, *malloc(), *calloc() ;

  if (n < 2)  message(531);  /* check # of arguments */

gp1:
  /* is identifier syntax valid ? */

  w = (bufp += WORDSIZ);
  if (namegood(w) != TRUE)  message(532);

  /* is the name already declared ? */

  switch(nameused(w))
  {
    case CONSTANT :  message(534);  break;
    case PARAMETER:  message(536);  break;
    case VARIABLE :  message(535);  break;
    default: /* name not declared */break;
  }

  /* allocate new slot in the "param" array */

  if (nparam >= NUMPARAM)  message(533);

  j = nparam ;  nparam++ ;

  if ((param[j] = (struct identifier *) malloc(IDENSIZ)) == NULL)
  message(504);

  strcpy(param[j]->nam,w) ;
  param[j]->itype = PARAMETER            ;
  param[j]->f     = NULL                 ;
  param[j]->valu  = 0.0                  ;
  param[j]->vmin  = param[j]->vmax = 0.0 ;

  if ((dom[j] = (domarr *) calloc(NUMDOM,DOMSIZ)) == NULL)  message(504);

  if (optlog == ON)
  {
    fprintf(logf,"[%d]%s = %f\n",j,param[j]->nam,param[j]->valu);
    fflush(logf);
  }

  bufp += WORDSIZ;
  if (strcmp(bufp,"")  != 0)
  if (strcmp(bufp,",") != 0)  message(531) ;
  else                        goto gp1     ;
}


/***************************************************************************
 *                                                                         *
 *  getdom -- get domain declaration & definition                          *
 *                                                                         *
 ***************************************************************************/

getdom (n,bufp)
int   n ;
char *bufp ;
{ char *w ;
  int   i, j, k, found, f1, f2 ;
  float x1, x2, x3, x4, x5, x6 ;

  if (n < 6)  message(541);  /* check # of arguments */

  /* 1st bracket value */

  bufp += WORDSIZ ;
  if ((i = getoken(bufp,&x1)) == 0) message(541);

dm0:  /* 1st bracket inequality */

  bufp += (i * WORDSIZ) ;
  if (strcmp(bufp,"<=") == 0)  { f1 = LESS_THAN + EQUAL_TO    ;  goto dm1 ; }
  if (strcmp(bufp,"<")  == 0)  { f1 = LESS_THAN               ;  goto dm1 ; }
  if (strcmp(bufp,">=") == 0)  { f1 = GREATER_THAN + EQUAL_TO ;  goto dm1 ; }
  if (strcmp(bufp,">" ) == 0)  { f1 = GREATER_THAN            ;  goto dm1 ; }

  message(541) ;

dm1:  /* is identifier syntax valid ? */

  w = (bufp += WORDSIZ);
  if (namegood(w) != TRUE)  message(532);

  /* locate position in "param" array */

  for (j = 0, found = OFF; found == OFF && j < nparam; )
  if (strcmp(bufp,param[j]->nam) == 0) found = ON; else j++;

  if (found == OFF)  message(543);

  /* 2nd bracket inequality */

  bufp += WORDSIZ ;
  if (strcmp(bufp,"<=") == 0)  { f2 = LESS_THAN + EQUAL_TO    ;  goto dm2 ; }
  if (strcmp(bufp,"<" ) == 0)  { f2 = LESS_THAN               ;  goto dm2 ; }
  if (strcmp(bufp,">=") == 0)  { f2 = GREATER_THAN + EQUAL_TO ;  goto dm2 ; }
  if (strcmp(bufp,">" ) == 0)  { f2 = GREATER_THAN            ;  goto dm2 ; }

  message(541) ;

dm2:  /* 2nd bracket value */

  bufp += WORDSIZ ;
  if ((i = getoken(bufp,&x2)) == 0) message(541);

dm3:  /* range test */

  /* filter out x1=x2, "<>" and "><" */


  if (x1 == x2)                           message(545);
  if ((f1 & f2 & ~EQUAL_TO) == OFF)  message(542);

  if ((f1 & f2 & ~EQUAL_TO) == LESS_THAN)
  {
    if (x1 < x2)  goto dm4 ;  /* inequality is logical */

    if ((f1 & f2 & EQUAL_TO) == EQUAL_TO)
    if (x1 <= x2) goto dm4 ;

    message(542);
  }
  else
  {
    if (x1 > x2)  goto dm4 ;   /* inequality is logical */

    if ((f1 & f2 & EQUAL_TO) == EQUAL_TO)
    if (x1 >= x2) goto dm4 ;

    message(542);
  }

dm4:  /* get trace value */

  bufp += (i * WORDSIZ) ;
  if (*bufp == '\0')  goto dma     ;
  if (*bufp !=  ',')  message(561) ;
  if ((i = getoken(bufp+=WORDSIZ,&x3)) != 0)  goto dm5 ;

dma:  /* apply default value for unit-step mode, 1% of domain      */
      /* if trace mode is unit-length, default is zero this point */

  i = 0;
  if (optulen == OFF)
  {
#if VER == LATTICE
    x3 = 0.01 *  abs(x1 - x2);
#endif
#if VER == UNIX
    x3 = 0.01 * fabs(x1 - x2);
#endif
  }
  else x3 = 0.0;

dm5:  /* get display value */
/*
  bufp += (i * WORDSIZ) ;
  if (*bufp == '\0')  goto dmb     ;
  if (*bufp !=  ',')  message(571) ;
  if ((i = getoken(bufp+=WORDSIZ,&x4)) != 0)  goto dm6 ;
*/
dmb:  /* apply default value, set to trace */

/*i = 0;*/
/*if (optulen == OFF)*/ x4 = x3;
/*else                  x4 = 0.0;*/

dm6:  /* get precision value */

  bufp += (i * WORDSIZ) ;
  if (*bufp == '\0')  goto dmc     ;
  if (*bufp !=  ',')  message(591) ;
  if ((i = getoken(bufp+=WORDSIZ,&x5)) != 0)  goto dm7 ;

dmc:  /* apply default value: 1% of trace */

  i = 0;
  if (optulen == OFF) x5 = 0.01 * x3;
  else                x5 = 0.0;

dm7:  /* get sensitivity value */

  bufp += (i * WORDSIZ) ;
  if (*bufp == '\0')  goto dmd      ;
  if (*bufp !=  ',')  message(592) ;
  if ((i = getoken(bufp+=WORDSIZ,&x6)) != 0)  goto dm8 ;

dmd:  /* apply default value: 0.001% of domain interval */

  i = 0;
#if VER == LATTICE
  x6 = 0.00001 *  abs(x1-x2);
#endif
#if VER == UNIX
  x6 = 0.00001 * fabs(x1-x2);
#endif

dm8:

  bufp += (i * WORDSIZ) ;
  if (*bufp != '\0')                       message(541) ;
  if ((int) param[j]->vmax >= NUMDOM)  message(544) ;

  k = (int) param[j]->vmax ;

  (*dom[j])[k].flag        = ((f2 & EQUAL_TO) << 4) + (f1 & EQUAL_TO)       ;
  (*dom[j])[k].dmin        =  (f1 & LESS_THAN) ? (double) x1 : (double) x2 ;
  (*dom[j])[k].dmax        =  (f1 & LESS_THAN) ? (double) x2 : (double) x1 ;

  (*dom[j])[k].trace       = (double) x3 ;
  (*dom[j])[k].display     = (double) x4 ;
  (*dom[j])[k].precision   = (double) x5 ;
  (*dom[j])[k].sensitivity = (double) x6 ;

  param[j]->itype |= DOMFLAG ;
  param[j]->vmax  += 1.0     ;

  if (optlog == ON)
  {
    fprintf(logf,"[%d]%s = ",j,param[j]->nam);
    fprintf(logf,"%c%f,",(f1 & EQUAL_TO)?'[':'(',(*dom[j])[k].dmin);
    fprintf(logf,"%f%c ", (*dom[j])[k].dmax,(f1 & EQUAL_TO)?']':')');
    fprintf(logf,"[%f]  ",(*dom[j])[k].trace       );
    fprintf(logf,"[%f]  ",(*dom[j])[k].display     );
    fprintf(logf,"[%f]  ",(*dom[j])[k].precision   );
    fprintf(logf,"[%f]\n",(*dom[j])[k].sensitivity);
    fflush(logf);
  }
}


/***************************************************************************
 *                                                                         *
 *  geteqt -- get defining equation                                        *
 *                                                                         *
 ***************************************************************************/

geteqt (n,bufp)
int   n ;
char *bufp ;
{ int j = 0, flag = OFF ;
  struct expr *f, *isform() ;

  if (n < 3)  message(581);  /* check # of arguments */

  /* check validity of identifier */

  while (flag == OFF && j < nvar)
  if (strcmp(bufp,var[j]->nam) == 0)  flag = ON;  else j++;

  if (flag == OFF)  message(582);

  /* check existence of '=' */

  if (*(bufp+=WORDSIZ) != '=')  message(581);

  /* check if variable already defined */

  if ((var[j]->itype & USEDFLAG) != OFF)  message(527);

  /* check if specified expression is valid */

  if ((f = isform(bufp+=WORDSIZ)) == NULL)  message(583);

  var[j]->itype |= USEDFLAG ;
  var[j]->f     = f        ;

  if (optlog == ON)
  {
    fprintf(logf,"[%d]%s\n",j,var[j]->nam);  fflush(logf);
  }
}


/***************************************************************************
 *                                                                         *
 *  getoken -- get a token                                                 *
 *                                                                         *
 ***************************************************************************/

getoken (bp,xp)
char  *bp;
float *xp;
```

```
{ float sign;
  int found, j, n = 0;

  if (*bp == '-')
  {
    sign = -1.0 ;  n++ ;  bp += WORDSIZ ;
  }
  else sign = 1.0 ;

  if (sscanf(bp,"%f",xp) != 0)
  {
    *xp *= sign ;  n++ ;  goto gt1 ;
  }

  for (j = 0, found = OFF; found == OFF && j < nconst; )
  if (strcmp(bp,const[j]->nam) == 0) found = ON; else j++;

  if (found != OFF)
  {
    *xp = sign * const[j]->valu ;  n++ ;
  }
  else n = 0 ;

gt1:

  return(n);
}


/*****************************************************************************
 *                                                                           *
 *  chkspec -- check misc. logic of plot specification                       *
 *                                                                           *
 *****************************************************************************/

chkspec ()
{ int i, j, k;
  int uconst = 0, uparam = 0, uvar = 0;
  struct identifier *tmp1;
  domarr *tmp2;
  double delta;

  /* constant check */

  for (j = 0; j < nconst; j++)
  if ((const[j]->itype & USEDFLAG) != OFF)  uconst += 1 ;

  /* parameter check */

  if (nparam <= 0)  message(601);

  for (j = 0; j < nparam; )
  if ((param[j]->itype & USEDFLAG) != OFF)
  {
    if ((param[j]->itype & DOMFLAG) == OFF)  message(603);
    uparam++ ;  j++ ;
```

```
}
else  /* not used in an equation; pull up one slot */
{
  for (i = j, tmp1 = param[j], tmp2 = dom[j]; i < nparam-1; i++)
  {
    param[i] = param[i+1];
    dom[i]   = dom[i+1];
  }
  param[nparam-1] = tmp1;  dom[nparam-1] = tmp2;
  nparam-- ;
}
if (optlog == ON) fprintf(logf,"upar=%2d | npar=%2d\n",uparam,nparam);
if (nparam > 2)  message(606);

/* variable check */

if (nvar <= 0)  message(605);

for (j = 0; j < nvar; )
if ((var[j]->itype & USEDFLAG) != OFF)
{
  if (var[j]->f == NULL)  message(604);
  uvar++ ;  j++ ;
}
else
{
  for (i = j, tmp1 = var[j]; i < nvar-1; i++) var[i] = var[i+1];
  var[nvar-1] = tmp1;
  nvar-- ;
}
if (optlog == ON) fprintf(logf,"uvar=%2d | nvar=%2d\n",uvar,nvar);
if (nvar > 3)  message(607);

if (nparam >= nvar)  message(602);

/* check basic requirements for tracing trace & display interval */

for (j = 0; j < nparam; j++)
for (k = 0; k < (int) param[j]->vmax; k++)
{
  if ((*dom[j])[k].trace < 0)
  {
    (*dom[j])[k].trace *= -1.0;         message(101);
  }
  if ((*dom[j])[k].display < 0)
  {
    (*dom[j])[k].display *= -1.0;       message(102);
  }
  if ((*dom[j])[k].precision < 0)
  {
    (*dom[j])[k].precision *= -1.0;     message(103);
  }
  if ((*dom[j])[k].sensitivity < 0)
  {
    (*dom[j])[k].sensitivity *= -1.0;  message(104);
  }
```

```
    if ((*dom[j])[k].trace == 0 && optulen == OFF)
    {
      (*dom[j])[k].trace = 0.01 * ((*dom[j])[k].dmax - (*dom[j])[k].dmin);
      message(105);
    }
    if ((*dom[j])[k].display < (*dom[j])[k].trace)
    {
      (*dom[j])[k].display = (*dom[j])[k].trace;
      message(106);
    }
  }

  /* adjusting domain bracket values */

  for (j = 0; j < nparam; j++)
  for (k = 0; k < (int) param[j]->vmax; k++)
  {
    delta = 0.001 * ((*dom[j])[k].dmax - (*dom[j])[k].dmin);
    if (((*dom[j])[k].flag & EQUAL_TO) == OFF)
      (*dom[j])[k].dmin += delta;

    if ((((*dom[j])[k].flag >> 4) & EQUAL_TO) == OFF)
      (*dom[j])[k].dmax -= delta;
  }

  /* calculate default unit length if tracing mode is unit length */

  if (optulen == ON) findulen();
}


/*****************************************************************************
 *                                                                           *
 *  namegood -- check identifier syntax                                      *
 *                                                                           *
 *****************************************************************************/

namegood (word)
char *word;
{ int j;

  /* is first character alphabet ? */

  if (isalpha(*word) == FALSE)  return(FALSE);

  /* is the rest alphanumeric */

  for (j = strlen(word)-1; j > 0; j--)
  if (isalnum(*(word+j)) == FALSE)
  {
    if (*(word+j) != '_')  return(FALSE);
  }

  /* is the name a keyword used by UGTRACE */

  if (isconstant(word)  != FALSE)  return(FALSE);
  if (isparameter(word) != FALSE)  return(FALSE);
  if (isdomain(word)    != FALSE)  return(FALSE);
  if (isvariable(word)  != FALSE)  return(FALSE);

  return(TRUE);
}


/***************************************************************************
 *                                                                         *
 *  nameused -- check identifier used or not                               *
 *                                                                         *
 ***************************************************************************/

nameused (word)
char *word;
{ int j;

  for (j = 0; j < nconst; j++)
  if (strcmp(word,const[j]->nam) == 0) return(CONSTANT);

  for (j = 0; j < nparam; j++)
  if (strcmp(word,param[j]->nam) == 0) return(PARAMETER);

  for (j = 0; j < nvar; j++)
  if (strcmp(word,var[j]->nam) == 0)   return(VARIABLE);

  return(0);
}


/***************************************************************************
 *                                                                         *
 *  isconstant -- check if given word is the keyword constant              *
 *                                                                         *
 ***************************************************************************/

isconstant(w)
char *w;
{
  if (isletter(*(w+0),'c') == FALSE)  return(FALSE);
  if (isletter(*(w+1),'o') == FALSE)  return(FALSE);
  if (isletter(*(w+2),'n') == FALSE)  return(FALSE);
  if (isletter(*(w+3),'s') == FALSE)  return(FALSE);
  if (isletter(*(w+4),'t') == FALSE)  return(FALSE);
  if (*(w+5) == '\0')  return(TRUE);

  if (isletter(*(w+5),'a') == FALSE)  return(FALSE);
  if (*(w+6) == '\0')  return(TRUE);

  if (isletter(*(w+6),'n') == FALSE)  return(FALSE);
  if (*(w+7) == '\0')  return(TRUE);

  if (isletter(*(w+7),'t') == FALSE)  return(FALSE);
  if (*(w+8) == '\0')  return(TRUE);
  return(FALSE);
```

```
}


/****************************************************************************
 *                                                                          *
 *  isparameter -- check if given word is the keyword parameter             *
 *                                                                          *
 ****************************************************************************/

isparameter(w)
char *w;
{
  if (isletter(*(w+0),'p') == FALSE)  return(FALSE);
  if (isletter(*(w+1),'a') == FALSE)  return(FALSE);
  if (isletter(*(w+2),'r') == FALSE)  return(FALSE);
  if (*(w+3) == '\0')  return(TRUE);

  if (isletter(*(w+3),'a') == FALSE)  return(FALSE);
  if (*(w+4) == '\0')  return(TRUE);

  if (isletter(*(w+4),'m') == FALSE)  return(FALSE);
  if (*(w+5) == '\0')  return(TRUE);

  if (isletter(*(w+5),'e') == FALSE)  return(FALSE);
  if (*(w+6) == '\0')  return(TRUE);

  if (isletter(*(w+6),'t') == FALSE)  return(FALSE);
  if (*(w+7) == '\0')  return(TRUE);

  if (isletter(*(w+7),'e') == FALSE)  return(FALSE);
  if (*(w+8) == '\0')  return(TRUE);

  if (isletter(*(w+8),'r') == FALSE)  return(FALSE);
  if (*(w+9) == '\0')  return(TRUE);
  return(FALSE);
}


/****************************************************************************
 *                                                                          *
 *  isvariable -- check if given word is the keyword variable               *
 *                                                                          *
 ****************************************************************************/

isvariable(w)
char *w;
{
  if (isletter(*(w+0),'v') == FALSE)  return(FALSE);
  if (isletter(*(w+1),'a') == FALSE)  return(FALSE);
  if (isletter(*(w+2),'r') == FALSE)  return(FALSE);
  if (*(w+3) == '\0')  return(TRUE);

  if (isletter(*(w+3),'i') == FALSE)  return(FALSE);
  if (*(w+4) == '\0')  return(TRUE);

  if (isletter(*(w+4),'a') == FALSE)  return(FALSE);
  if (*(w+5) == '\0')  return(TRUE);

  if (isletter(*(w+5),'b') == FALSE)  return(FALSE);
  if (*(w+6) == '\0')  return(TRUE);

  if (isletter(*(w+6),'l') == FALSE)  return(FALSE);
  if (*(w+7) == '\0')  return(TRUE);

  if (isletter(*(w+7),'e') == FALSE)  return(FALSE);
  if (*(w+8) == '\0')  return(TRUE);
  return(FALSE);
}


/*****************************************************************************
 *                                                                           *
 *  isdomain -- check if given word is the keyword domain                    *
 *                                                                           *
 *****************************************************************************/

isdomain(w)
char *w;
{
  if (isletter(*(w+0),'d') == FALSE)  return(FALSE);
  if (isletter(*(w+1),'o') == FALSE)  return(FALSE);
  if (isletter(*(w+2),'m') == FALSE)  return(FALSE);
  if (*(w+3) == '\0')  return(TRUE);

  if (isletter(*(w+3),'a') == FALSE)  return(FALSE);
  if (*(w+4) == '\0')  return(TRUE);

  if (isletter(*(w+4),'i') == FALSE)  return(FALSE);
  if (*(w+5) == '\0')  return(TRUE);

  if (isletter(*(w+5),'n') == FALSE)  return(FALSE);
  if (*(w+6) == '\0')  return(TRUE);
  return(FALSE);
}


/*****************************************************************************
 *                                                                           *
 *  isletter -- match letter                                                 *
 *                                                                           *
 *****************************************************************************/

isletter(c1,c2)
char c1, c2;
{ int bool;

  if (isascii(c1) == FALSE)
  {
    bool = FALSE;
  }
  else if (isupper(c1) == FALSE)
  {
    bool = (c1 == c2) ? TRUE : FALSE;
  }
  else
  {
    bool = (tolower(c1) == c2) ? TRUE : FALSE;
  }
  return(bool);
}


/**************************************************************************
 *                                                                        *
 *  findulen -- find default unit length for each domain                  *
 *                                                                        *
 **************************************************************************/

findulen()
{ int k;
  double evalu();
  double x0, y0, z0;
  double x1, y1, z1;

  if (nparam == 2) goto surface;

curve:

  for (k = 0; k < (int)param[0]->vmax; k++)
  {
    if ((*dom[0])[k].trace == 0.0)
    {
      param[0]->valu = (*dom[0])[k].dmin;
      x0 = evalu(var[0]->f);
      y0 = evalu(var[1]->f);

      param[0]->valu += 0.01 * ((*dom[0])[k].dmax - (*dom[0])[k].dmin);
      x1 = evalu(var[0]->f);
      y1 = evalu(var[1]->f);

      (*dom[0])[k].trace = (*dom[0])[k].display
      = sqrt( (x1-x0)*(x1-x0) + (y1-y0)*(y1-y0) );
    }
    if ((*dom[0])[k].precision == 0.0)
    (*dom[0])[k].precision = 0.0001 * (*dom[0])[k].trace;
  }
  return;

surface:

  param[1]->valu = (*dom[1])[0].dmin;  /* set v to minimum */

  for (k = 0; k < (int)param[0]->vmax; k++)
  {
    if ((*dom[0])[k].trace == 0.0)
    {
      param[0]->valu = (*dom[0])[k].dmin;
      x0 = evalu(var[0]->f);
      y0 = evalu(var[1]->f);
      z0 = evalu(var[2]->f);

      param[0]->valu += 0.01 * ((*dom[0])[k].dmax - (*dom[0])[k].dmin);
      x1 = evalu(var[0]->f);
      y1 = evalu(var[1]->f);
      z1 = evalu(var[2]->f);

      (*dom[0])[k].trace = (*dom[0])[k].display
      = sqrt( (x1-x0)*(x1-x0) + (y1-y0)*(y1-y0) + (z1-z0)*(z1-z0) );
    }
    if ((*dom[0])[k].precision == 0.0)
    (*dom[0])[k].precision = 0.0001 * (*dom[0])[k].trace;
  }

  param[0]->valu = (*dom[0])[0].dmin;  /* set u to minimum */

  for (k = 0; k < (int)param[1]->vmax; k++)
  {
    if ((*dom[1])[k].trace == 0.0)
    {
      param[1]->valu = (*dom[1])[k].dmin;
      x0 = evalu(var[0]->f);
      y0 = evalu(var[1]->f);
      z0 = evalu(var[2]->f);

      param[1]->valu += 0.01 * ((*dom[1])[k].dmax - (*dom[1])[k].dmin);
      x1 = evalu(var[0]->f);
      y1 = evalu(var[1]->f);
      z1 = evalu(var[2]->f);

      (*dom[1])[k].trace = (*dom[1])[k].display
      = sqrt( (x1-x0)*(x1-x0) + (y1-y0)*(y1-y0) + (z1-z0)*(z1-z0) );
    }
    if ((*dom[1])[k].precision == 0.0)
    (*dom[1])[k].precision = 0.0001 * (*dom[1])[k].trace;
  }
  return;
}
```

```
/***************************************************************************
 *                                                                         *
 *  expr.c -- misc functions for expression parsing                        *
 *                                                                         *
 *  Author: Henri Cheung at McMaster University, Hamilton, Ontario         *
 *                                                                         *
 *                                                                         *
 *  grammer used:                                                          *
 *                                                                         *
 *  <form>    ::= <expr> "\0"                                              *
 *  <expr>    ::= <term> | <term><addop><expr>                             *
 *  <term>    ::= <fact1> | <fact1><multop><term>                          *
 *  <fact1>   ::= <func><fact2> | <fact2>                                  *
 *  <fact2>   ::= <argu> | <argu><powop><fact2>                            *
 *  <argu>    ::= <lit> | <iden> | "("<expr>")"                            *
 *  <iden>    ::= <string> | "-"<string>                                   *
 *  <lit>     ::= <real> | <int>                                           *
 *  <addop>   ::= "+" | "-"                                                *
 *  <multop>  ::= "*" | "/"                                                *
 *  <func>    ::= "sin"   | "cos"   | "tan"   |                            *
 *                "asin"  | "acos"  | "atan"  |                            *
 *                "sinh"  | "cosh"  | "tanh"  |                            *
 *                "asinh" | "acosh" | "atanh" |    [ Unix only ]           *
 *                "exp"   | "log"   | "log10" |                            *
 *                "sqrt"                                                   *
 *                                                                         *
 *  <powop>   ::= "^"                                                      *
 *                                                                         *
 ***************************************************************************/


#include  <stdio.h>
#include  <ctype.h>
#include  "global.h"
#include  "struct.h"

extern FILE *logf ;
extern int  optlog;
extern int nconst, nparam;
extern struct identifier *const[], *param[];

char *bufp;


/***************************************************************************
 *                                                                         *
 *  isform -- is formula ?                                                 *
 *                                                                         *
 ***************************************************************************/

struct expr *
isform (buffer)
char *buffer;
{ struct expr *isexpr(), *f = NULL;

  if (*(bufp = buffer) == '\0')  return(NULL);
  if ((f = isexpr()) == NULL)    return(NULL);

  if (*bufp == '\0')             return(f);

  return(NULL);
}


/**************************************************************************
 *                                                                        *
 *  isexpr -- is expression ?                                             *
 *                                                                        *
 **************************************************************************/

struct expr *
isexpr ()
{ struct expr *isaddop(), *isterm(), *e = NULL, *a = NULL;

  if (*bufp == '\0')            return(NULL);

  if ((e = isterm()) == NULL)   return(NULL);

  while (0 == 0)
  {
  if ((a = isaddop()) == NULL)  return(e);
  if ((a->r = isterm()) != NULL)
  {
    a->l = e;  e = a ;
  }
  else  message(553);
  }
}


/**************************************************************************
 *                                                                        *
 *  isterm -- is term ?                                                   *
 *                                                                        *
 **************************************************************************/

struct expr *
isterm ()
{ struct expr *isfactl(), *ismultop(), *t = NULL, *m = NULL;

  if (*bufp == '\0')             return(NULL);

  if ((t = isfactl()) == NULL)   return(NULL);

  while (0 == 0)
  {
  if ((m = ismultop()) == NULL)  return(t);
  if ((m->r = isfactl()) != NULL)
  {
    m->l = t;  t = m;
  }
  else  message(554);
  }
}


/************************************************************************
 *                                                                      *
 *  isfact1 -- is 1st level factor ?                                    *
 *                                                                      *
 ************************************************************************/

struct expr *
isfact1 ()
{ struct expr *isfunc(), *isfact2(), *f = NULL;

  if (*bufp == '\0')  return(NULL);

  if ((f = isfunc()) != NULL)
  if ((f->r = isfact2()) != NULL)  return(f);
  else                             message(554);

  if ((f = isfact2()) != NULL)     return(f);
  else                             return(NULL);
}


/************************************************************************
 *                                                                      *
 *  isfact2 -- is 2nd level factor ?                                    *
 *                                                                      *
 ************************************************************************/

struct expr *
isfact2 ()
{ struct expr *isargu(), *ispowop(), *f = NULL, *p = NULL;

  if (*bufp == '\0')             return(NULL);

  if ((f = isargu()) == NULL)    return(NULL);

  while (0 == 0)
  {
  if ((p = ispowop()) == NULL)  return(f);
  if ((p->r = isargu()) != NULL)
  {
      p->l = f;  f = p;
  }
  else  message(554);
  }
}


/************************************************************************
 *                                                                      *
 *  isargu -- is argument ?                                             *
 *                                                                      *
 *****************************************************************************/

struct expr *
isargu ()
{ struct expr *isexpr(), *iside(), *islit(),  *a = NULL;

  if (*bufp == '\0')            return(NULL);

  if ((a = islit()) != NULL)  return(a);

  if (*bufp == '(')
  {
    bufp += WORDSIZ;
    if ((a = isexpr()) != NULL)
      if (*bufp == ')')
      {
        bufp += WORDSIZ;        return(a);
      }
      else                      message(557);
    else                        message(552);
  }

  if ((a = isiden()) != NULL)  return(a);

  return(NULL);
}


/*****************************************************************************
 *                                                                           *
 *  isiden -- is identifier ?                                                *
 *                                                                           *
 *****************************************************************************/

struct expr *
isiden ()
{ float sign;
  int j, found = 0;
  struct expr *i = NULL;
  char *localp, s[WORDSIZ], *malloc();

  localp = bufp;

  if (*localp == '\0')  return(NULL);

  if (*localp == '-')
  {
    sign = -1.0;  localp += WORDSIZ;
  }
  else sign = 1.0;

  if (*localp == '\0')  return(NULL);

  /* check validity of identifier syntax */

  if (namegood(localp) != TRUE)  return(NULL);
  /* check if it is a defined constant */

  j = 0;
  while (found == 0 && j < nconst)
  if (strcmp(localp,const[j]->nam) == 0)  found = CONSTID;  else j++;

  if (found == CONSTID)  { const[j]->itype |= USEDFLAG ;  goto id1; }

  /* check if it is a defined parameter */

  j = 0;
  while (found == 0 && j < nparam)
  if (strcmp(localp,param[j]->nam) == 0)  found = PARAMID;  else j++;

  if (found == PARAMID)  { param[j]->itype |= USEDFLAG ;  goto id1; }

  message(556) ;

id1:

  if ((i = (struct expr *) malloc(EXPRSIZ)) == NULL)  message(504);
  if (found == PARAMID)
  {
    i->ctype = PARAMID;
    i->info.ptr = param[j] ;
  }
  else
  {
    i->ctype = REAL;
    i->info.rval = sign * const[j]->valu;
  }
  i->l = i->r = NULL ;

  if (sign < 0)  bufp += WORDSIZ;
  bufp += WORDSIZ;

  return(i);
}


/******************************************************************************
 *                                                                            *
 *  islit -- is literal ?                                                     *
 *                                                                            *
 ******************************************************************************/

struct expr *
islit ()
{ int i ;
  float x, sign ;
  char *localp, *malloc() ;
  struct expr *l = NULL;

  localp = bufp;
  if (*localp == '\0')  return(NULL);

  if (*localp == '-')
  {
    sign = -1.0;  localp += WORDSIZ;
  }
  else sign = 1.0;

  if (*localp == '\0')  return(NULL);

  if (sscanf(localp,"%f",&x) == 1)
  {
    if ((l = (struct expr *) malloc(EXPRSIZ)) == NULL)  message(504);
    l->ctype = REAL;
    l->info.rval = (double) (sign * x);
    l->l = l->r = NULL ;

    if (sign < 0)  bufp += WORDSIZ;
    bufp += WORDSIZ;
  }
  else if (sscanf(localp,"%d",&i) == 1)
  {
    if ((l = (struct expr *) malloc(EXPRSIZ)) == NULL)  message(504);
    l->ctype = INTEGER;
    l->info.ival = ((int) sign) * i;
    l->l = l->r = NULL ;

    if (sign < 0)  bufp += WORDSIZ;
    bufp += WORDSIZ;
  }

  return(l);
}


/*****************************************************************************
 *                                                                           *
 *  isfunc -- is function ?                                                  *
 *                                                                           *
 *****************************************************************************/

struct expr *
isfunc ()
{ int flag = 0 ;
  char *malloc() ;
  struct expr *f = NULL ;

  if (*bufp == '\0')  return(NULL);

  if (strcmp(bufp,"cos"  ) == 0)  { flag++;  goto fn1; }
  if (strcmp(bufp,"sin"  ) == 0)  { flag++;  goto fn1; }
  if (strcmp(bufp,"tan"  ) == 0)  { flag++;  goto fn1; }
  if (strcmp(bufp,"acos" ) == 0)  { flag++;  goto fn1; }
  if (strcmp(bufp,"asin" ) == 0)  { flag++;  goto fn1; }
  if (strcmp(bufp,"atan" ) == 0)  { flag++;  goto fn1; }
  if (strcmp(bufp,"cosh" ) == 0)  { flag++;  goto fn1; }
  if (strcmp(bufp,"sinh" ) == 0)  { flag++;  goto fnl; }
  if (strcmp(bufp,"tanh" ) == 0)  { flag++;  goto fnl; }
#if VER == UNIX  /* Unix version only */
  if (strcmp(bufp,"acosh") == 0)  { flag++;  goto fnl; }
  if (strcmp(bufp,"asinh") == 0)  { flag++;  goto fnl; }
  if (strcmp(bufp,"atanh") == 0)  { flag++;  goto fnl; }
#endif
  if (strcmp(bufp,"exp"  ) == 0)  { flag++;  goto fnl; }
  if (strcmp(bufp,"log"  ) == 0)  { flag++;  goto fnl; }
  if (strcmp(bufp,"log10") == 0)  { flag++;  goto fnl; }
  if (strcmp(bufp,"sqrt" ) == 0)  { flag++;  goto fnl; }

fnl:

  if (flag != 0)
  {
    if ((f = (struct expr *) malloc(EXPRSIZ)) == NULL)  message(504);
    strcpy(f->info.sval,bufp);
    f->ctype = FUNCTION;
    f->l = f->r = NULL ;
    bufp += WORDSIZ;
  }
  return(f);
}


/***********************************************************************
 *                                                                     *
 *  isaddop -- is addition operator ?                                  *
 *                                                                     *
 ***********************************************************************/

struct expr *
isaddop ()
{ char *malloc() ;
  struct expr *a = NULL ;

  if (*bufp == '\0')  return(NULL);

  if (strcmp(bufp,"+") == 0 || strcmp(bufp,"-") == 0)
  {
    if ((a = (struct expr *) malloc(EXPRSIZ)) == NULL)  message(504);
    strcpy(a->info.sval,bufp);
    a->ctype = OPERATOR;
    a->l = a->r = NULL ;
    bufp += WORDSIZ;
  }
  return(a);
}


/***********************************************************************
 *                                                                     *
 *  ismultop -- is multiplication operator ?                           *
 *                                                                     *
 ***********************************************************************/
```

```
struct expr *
ismultop ()
{ char *malloc() ;
  struct expr *m = NULL ;

  if (*bufp == '\0')  return(NULL);

  if (strcmp(bufp,"*") == 0 || strcmp(bufp,"/") == 0)
  {
    if ((m = (struct expr *) malloc(EXPRSIZ)) == NULL)  message(504);
    strcpy(m->info.sval,bufp);
    m->ctype = OPERATOR;
    m->l = m->r = NULL ;
    bufp += WORDSIZ;
  }
  return(m);
}


/***********************************************************************
 *                                                                     *
 *  ispowop -- is power operator ?                                     *
 *                                                                     *
 ***********************************************************************/

struct expr *
ispowop ()
{ char *malloc() ;
  struct expr *p = NULL ;

  if (*bufp == '\0')  return(NULL);

  if (strcmp(bufp,"^") == 0)
  {
    if ((p = (struct expr *) malloc(EXPRSIZ)) == NULL)  message(504);
    strcpy(p->info.sval,bufp);
    p->ctype = OPERATOR;
    p->l = p->r = NULL ;
    bufp += WORDSIZ;
  }
  return(p);
}


/***********************************************************************
 *                                                                     *
 *  reclaim -- release memory used by and expression                   *
 *                                                                     *
 ***********************************************************************/

struct expr *
reclaim(f)
struct expr *f ;
{
  if (f == NULL)  return(NULL);
  reclaim(f->l);
  reclaim(f->r);

#if VER == LATTICE  /* Lattice C Version */
  if (free((char *) f) ==   0)  message(505);
#endif
#if VER == UNIX     /* Unix version      */
  if (free((char *) f) == 239)  message(505);
#endif
}


/*******************************************************************************
 *                                                                             *
 *  parsef -- parse an expression (auditing)                                   *
 *                                                                             *
 *******************************************************************************/

parsef (n,f)
int n;
struct expr *f ;
{
  if (f != NULL)
  {
    parsef(n+1,f->l);
    switch(f->ctype)
    {
      case  1: printf("i|%d|level %d\n",f->info.ival,n);       break;
      case  2: printf("r|%f|level %d\n",f->info.rval,n);       break;
      case  8: printf("s|%s|level %d\n",f->info.sval,n);       break;
      case 16: printf("s|%s|level %d\n",f->info.sval,n);       break;
      default: printf("s|%f|level %d\n",f->info.ptr->valu,n);  break;
    }
    parsef(n+1,f->r);
  }
}
```

```
/***************************************************************************
 *                                                                         *
 *  evalu.c -- evalu an expression tree built according to parse format   *
 *                                                                         *
 *  Author: Henri Cheung at McMaster University, Hamilton, Ontario        *
 *                                                                         *
 ***************************************************************************/


#include  <stdio.h>
#include  <math.h>
#include  "global.h"
#include  "struct.h"
#include  "evalu.h"

extern FILE *logf ;
extern int  optlog;


/***************************************************************************
 *                                                                         *
 *  evalu -- evalu an expression tree built according to parse format     *
 *                                                                         *
 ***************************************************************************/

double
evalu (f)
struct expr *f ;
{ int flag = 0;
  double result, evalu2();

  result = evalu2(f,&flag);
  if (flag > 0) message(703);
  return(result);
}


/***************************************************************************
 *                                                                         *
 *  evalu2 -- evalu an expression tree built according to parse format    *
 *                                                                         *
 ***************************************************************************/

double
evalu2 (f,flag)
int *flag;
struct expr *f ;
{ double a1, a2, evalu2() ;

  if (f == NULL)  { *flag |= ENUL;  return(0.0); }

  switch(f->ctype)
  {
    case INTEGER: return((double)f->info.ival);  break;
    case REAL    : return(         f->info.rval);  break;
    case OPERATOR:
      a1 = evalu2(f->l,flag);  a2 = evalu2(f->r,flag);
      if (strcmp(f->info.sval,"+") == 0)            return(a1+a2);
      if (strcmp(f->info.sval,"-") == 0)            return(a1-a2);
      if (strcmp(f->info.sval,"*") == 0)            return(a1*a2);
      if (strcmp(f->info.sval,"/") == 0)
      {
        if (a2 == 0) { *flag |= EZERO;  a2 = 1; }    return(a1/a2);
      }
      if (strcmp(f->info.sval,"^") == 0)
      {
/*      if (a1 < 0) { *flag |= ENEGP;  a2 = ceil(a2); }  return(pow(a1,a2));*/
        if (a1 < 0)   a2 = ceil(a2);                     return(pow(a1,a2));
      }
      break;
    case FUNCTION:
      a2 = evalu2(f->r,flag);
      if (strcmp(f->info.sval,"cos")   == 0)  return(cos(a2)  );
      if (strcmp(f->info.sval,"sin")   == 0)  return(sin(a2)  );
      if (strcmp(f->info.sval,"tan")   == 0)  return(tan(a2)  );
      if (strcmp(f->info.sval,"acos")  == 0)  return(acos(a2) );
      if (strcmp(f->info.sval,"asin")  == 0)  return(asin(a2) );
      if (strcmp(f->info.sval,"atan")  == 0)  return(atan(a2) );
      if (strcmp(f->info.sval,"cosh")  == 0)  return(cosh(a2) );
      if (strcmp(f->info.sval,"sinh")  == 0)  return(sinh(a2) );
      if (strcmp(f->info.sval,"tanh")  == 0)  return(tanh(a2) );
#if VER == UNIX  /* Unix version only */
      if (strcmp(f->info.sval,"acosh") == 0)  return(acosh(a2));
      if (strcmp(f->info.sval,"asinh") == 0)  return(asinh(a2));
      if (strcmp(f->info.sval,"atanh") == 0)  return(atanh(a2));
#endif
      if (strcmp(f->info.sval,"exp")   == 0)  return(exp(a2)  );
      if (strcmp(f->info.sval,"log")   == 0)  return(log(a2)  );
      if (strcmp(f->info.sval,"log10") == 0)  return(log10(a2));
      if (strcmp(f->info.sval,"sqrt")  == 0)  return(sqrt(a2) );
      break;
    case CONSTID:
    case PARAMID:
    case VARID  : return(f->info.ptr->valu);  break;
    default :     *flag |= 256;  return(0);   break;
  }
}
```

```
/**************************************************************************
 *                                                                        *
 *  unitstep.c -- functions used for unit step plotting                  *
 *                                                                        *
 *  Author: Henri Cheung at McMaster University, Hamilton, Ontario       *
 *                                                                        *
 **************************************************************************/


#include  <stdio.h>
#include  <math.h>
#include  "global.h"
#include  "struct.h"
#include  "struct2.h"

extern FILE *logf;
extern int optlog, optulen;
extern int nconst, nparam, nvar;
extern struct identifier *const[], *param[], *var[];
extern struct domain (*dom[])[NUMDOM];

struct curve   *cur2d = NULL;
struct curve   *cur3d = NULL;
struct surface *sur3d = NULL;


/**************************************************************************
 *                                                                        *
 *  plotspec -- lead module of plot shell                                 *
 *                                                                        *
 **************************************************************************/

double
plotspec()
{ double t = 0.0;
  struct curve   *curc, *plotcurves();
  struct surface *surf, *plotsurfaces();

  /* is it parametric ? */


  /* is it 2-D ? */

  if (nvar == 2)          cur2d = plotcurves(0) ;
  else if (nparam == 1)   cur3d = plotcurves(0) ;
  else                    sur3d = plotsurfaces();

  if (nvar == 2)
  {
    for (curc = cur2d;  curc != NULL;  curc = curc->nc)  t += curc->nump;

    if (optlog == ON)
    for (curc = cur2d;  curc != NULL;  curc = curc->nc)  logcurve(curc,2);
  }
  else if (nparam == 1)
  {
    for (curc = cur3d;  curc != NULL;  curc = curc->nc)  t += curc->nump;

    if (optlog == ON)
    for (curc = cur3d;  curc != NULL;  curc = curc->nc)  logcurve(curc,3);
  }
  else
  {
    for (surf = sur3d;  surf != NULL;  surf = surf->ns)
    for (curc = surf->root;  curc != NULL;  curc = curc->nc)
    {
      surf->nump += curc->nump;  t += curc->nump;
    }
    if (optlog == ON)
    for (surf = sur3d; surf != NULL; surf = surf->ns)  logsurface(surf);
  }

  return(t);
}


/**************************************************************************
 *                                                                        *
 *  plotsurfaces -- 3D, 2 parameter @ param[0] & param[1], 3 variables     *
 *                                                                        *
 **************************************************************************/

struct surface *
plotsurfaces()
{ int k0 = 0, k1 = 0;
  struct surface *first, *surf, *uslsurface(), *ullsurface();

  first = (optulen == ON) ? ullsurface(k0,k1) : uslsurface(k0,k1) ;

  if (first == NULL)  goto sx;

  surf = first;

s1:  /* checking exit conditions */

  if ((++k0) < (int)param[0]->vmax)  goto s2;  else  k0 = 0;
  if ((++k1) < (int)param[1]->vmax)  goto s2;  else  goto sx;

s2:  /* for current combination of k0 & k1 */

  surf->ns = (optulen == ON) ? ullsurface(k0,k1) : uslsurface(k0,k1) ;

  if (surf->ns == NULL)
  {
    surf->sflag = ERRMEM;  goto sx;
  }

  surf = surf->ns;
  goto s1;

sx:  /* exit point */
```

```
  return(first);
}


/*****************************************************************************
 *                                                                           *
 *  uslsurface -- plot one surface                                           *
 *                                                                           *
 *****************************************************************************/

struct surface *
uslsurface(k0,k1)
int k0, k1;
{ int quit = FALSE;
  struct curve *curc, *uslcurve();
  struct surface *surf, *mksurface();

  if ((surf = mksurface(k0,k1)) == NULL)  goto ps2;

  /* find first curve */

  param[1]->valu = (*dom[1])[k1].dmin ;

  if ((surf->root = uslcurve(0,k0)) == NULL)
  {
    surf->sflag = ERRMEM;  goto ps2;
  }
  surf->numc += 1.0;
  curc = surf->root;

ps1:  /* next curve */

  if ((param[1]->valu += (*dom[1])[k1].trace) >= (*dom[1])[k1].dmax)
  {
    quit = TRUE;
    if (param[1]->valu > (*dom[1])[k1].dmax)
    param[1]->valu = (*dom[1])[k1].dmax;
  }

  if ((curc->nc = uslcurve(0,k0)) == NULL)
  {
    curc->cflag = ERRMEM;  goto ps2;
  }
  surf->numc += 1.0;
  curc = curc->nc  ;

  if (quit == TRUE)  goto ps2;

  goto ps1;

ps2:  /* exit point */

  return(surf);
}
```

```
/***************************************************************************
 *                                                                         *
 *  plotcurves -- 2D or 3D, 1 parameter @ param[j], 2 or 3 variables       *
 *                                                                         *
 ***************************************************************************/

struct curve *
plotcurves(j)
int j;
{ int k = 0;
  struct curve *first, *curc, *uslcurve(), *ullcurve();

  /* plot first domain */

  first = (optulen == ON) ? ullcurve(j,k) : uslcurve(j,k);
  if (first == NULL)  goto cx;

  /* next domain if required, jth parameter, kth domain */

  for (curc = first, k = 1;  k < (int) param[j]->vmax;  curc = curc->nc, k++)
  {
    curc->nc = (optulen == ON) ? ullcurve(j,k) : uslcurve(j,k);
    if (curc->nc == NULL)
    {
      curc->cflag = ERRMEM;
      goto cx;
    }
  }

cx:  /* exit point */

 return(first);
}


/***************************************************************************
 *                                                                         *
 *  uslcurve -- 2D or 3D, 1 parameter @ param[j], 2 or 3 variables         *
 *                                                                         *
 ***************************************************************************/

struct curve *
uslcurve(j,k)
int j, k ;
{ double evalu() ;
  int i, quit = FALSE ;
  struct point *curp, *mkpoint() ;
  struct curve *curc, *mkcurve() ;

  if ((curc = mkcurve(j,k)) == NULL) goto pc2;

  /* for current domain range, set to left bracket value */

  param[j]->valu = (*dom[j])[k].dmin;
  for (i = 0;  i < nvar;  i++) var[i]->valu = evalu(var[i]->f) ;
  /* find 1st point in current domain */

  if ((curc->head = mkpoint()) == NULL)
  {
    curc->cflag = ERRMEM;  goto pc2;
  }
  curc->nump += 1.0;
  curp = curc->head;

pc1:  /* find next point in current domain range */

  if ((param[j]->valu += (*dom[j])[k].trace) >= (*dom[j])[k].dmax)
  {
    quit = TRUE;
    if (param[j]->valu > (*dom[j])[k].dmax)
    param[j]->valu = (*dom[j])[k].dmax;
  }

  for (i = 0;  i < nvar;  i++)  var[i]->valu = evalu(var[i]->f);

  if ((curp->np = mkpoint()) == NULL)
  {
    curp->pflag = ERRMEM;  goto pc2;
  }
  curc->nump += 1.0;
  curp = curp->np  ;

  if (quit == TRUE)  goto pc2;

  goto pc1;

pc2:  /* exit point */

  return(curc);
}


/*****************************************************************************
 *                                                                           *
 *  mkpoint -- allocate a point record and return its pointer                *
 *                                                                           *
 *****************************************************************************/

struct point *
mkpoint()
{ int  i ;
  char *malloc() ;
  struct point *p = NULL ;

  if ((p = (struct point *) malloc(PTSIZ)) == NULL)  goto ppx;

  p->np = NULL ;
  p->pflag = 0 ;

  for (i = 0;  i < PARMAX;  i++)
  p->u[i] = (i < nparam) ? param[i]->valu : 0.0 ;
  for (i = 0;  i < VARMAX;  i++)
  p->x[i] = (i < nvar) ? var[i]->valu : 0.0 ;

  for (i = 0;  i < nvar  ;  i++)
  if       (var[i]->vmin > var[i]->valu)  var[i]->vmin = var[i]->valu ;
  else if (var[i]->valu > var[i]->vmax)  var[i]->vmax = var[i]->valu ;

ppx:

  return(p);
}


/****************************************************************************
 *                                                                          *
 *  mkcurve -- allocate record for a new curve                              *
 *                                                                          *
 ****************************************************************************/

struct curve *
mkcurve(a,b)
int a, b;
{ char *malloc();
  struct curve *c = NULL;

  if ((c = (struct curve *) malloc(CURVESIZ)) == NULL)  goto ncx;

  c->cflag = 0 ;
  c->i1    = a ;     /* ath parameter */
  c->i2    = b ;     /* bth domain    */
  c->nump  = 0.0  ;
  c->head  = NULL ;  /* pointer to 1st point  */
  c->nc    = NULL ;  /* pointer to next curve */

ncx:

  return(c);
}


/****************************************************************************
 *                                                                          *
 *  mksurface -- allocate record for a new surface                          *
 *                                                                          *
 ****************************************************************************/

struct surface *
mksurface(a,b)
int a, b;
{ char *malloc();
  struct surface *s = NULL;

  if ((s = (struct surface *) malloc(SURFASIZ)) == NULL)  goto nsx;

  s->sflag = 0 ;
  s->i1     = a ;     /* ath domain of 0th parameter */
  s->i2     = b ;     /* bth domain of 1st parameter */
  s->nump   = 0.0   ;
  s->numc   = 0.0   ;
  s->root   = NULL ;  /* pointer to 1st curve    */
  s->ns     = NULL ;  /* pointer to next surface */

nsx:

  return(s);
}
```

```
/*************************************************************************
 *                                                                       *
 *  unitlen.c -- functions used for unit length plotting                 *
 *                                                                       *
 *  Author: Henri Cheung at McMaster University, Hamilton, Ontario       *
 *                                                                       *
 *************************************************************************/


#include  <stdio.h>
#include  <math.h>
#include  "global.h"
#include  "struct.h"
#include  "struct2.h"

extern FILE *logf ;
extern int  optlog;
extern int nconst, nparam, nvar;
extern struct identifier *const[], *param[], *var[];
extern struct domain (*dom[])[NUMDOM];


/*************************************************************************
 *                                                                       *
 *  ullsurface --                                                        *
 *                                                                       *
 *************************************************************************/

struct surface *
ullsurface(k0,k1)
int k0, k1;
{ int quit = FALSE;
  double dv, homein();
  struct curve *curc, *ullcurve();
  struct surface *surf, *mksurface();

  if ((surf = mksurface(k0,k1)) == NULL)  goto us2;

  /* set parameter(s) to left bracket value of current combination */

  param[1]->valu = (*dom[1])[k1].dmin ;

  /* first curve */

  if ((surf->root = ullcurve(0,k0)) == NULL)
  {
    surf->sflag = ERRMEM;  goto us2;
  }
  surf->numc += 1.0;
  curc = surf->root;

  /* set trial dv to tracing interval (starting point already plotted) */

  dv = 0.01 * ((*dom[1])[k1].dmax - (*dom[1])[k1].dmin);

us1:  /* find next curve */
```

```
  param[0]->valu = (*dom[0])[k0].dmin;

  if ((dv = homein(curc->head,1,k1,param[1]->valu,dv)) <= 0.0)  goto us2;

  if (param[1]->valu >= (*dom[1])[k1].dmax)
  {
    quit = TRUE;
    if (param[1]->valu > (*dom[1])[k1].dmax)
    param[1]->valu = (*dom[1])[k1].dmax;
  }

  if ((curc->nc = ullcurve(0,k0)) == NULL)
  {
    curc->cflag = ERRMEM;  goto us2;
  }
  surf->numc += 1.0;
  curc = curc->nc;

  if (quit == TRUE)  goto us2;

  if (dv < (*dom[1])[k1].sensitivity)  { message(201);  goto us2; }

  goto us1;

us2:  /* exit point */

  return(surf);
}


/***************************************************************************
 *                                                                         *
 *  ullcurve -- 2D or 3D, 1 parameter @ param[j], 2 or 3 variables         *
 *                                                                         *
 ***************************************************************************/

struct curve *
ullcurve(j,k)
int j, k;
{ int i, quit = FALSE;
  double du, homein(), evalu();
  struct point *curp, *mkpoint();
  struct curve *curc, *mkcurve();

  if ((curc = mkcurve(j,k)) == NULL) goto uc2;

  param[j]->valu = (*dom[j])[k].dmin;
  for (i = 0;  i < nvar;  i++) var[i]->valu = evalu(var[i]->f);

  /* find 1st point in current domain */

  if ((curc->head = mkpoint()) == NULL)
  {
    curc->cflag = ERRMEM;  goto uc2;
  }
  curc->nump += 1.0;
  curp = curc->head;

  /* set trial du to tracing interval */

  du = 0.01 * ((*dom[j])[k].dmax - (*dom[j])[k].dmin);

uc1:  /* find next point in curp domain range */

  if ((du = homein(curp,j,k,param[j]->valu,du)) <= 0.0)  goto uc2;

  if (param[j]->valu > (*dom[j])[k].dmax)
  {
    quit = TRUE;
    if (param[j]->valu >= (*dom[j])[k].dmax)
      param[j]->valu = (*dom[j])[k].dmax;
    for (i = 0; i < nvar; i++) var[i]->valu = evalu(var[i]->f);
  }

  if ((curp->np = mkpoint()) == NULL)
  {
    curp->pflag = ERRMEM;  goto uc2;
  }
  curc->nump += 1.0;
  curp = curp->np;

  if (quit == TRUE)  goto uc2;

  if (du < (*dom[j])[k].sensitivity)  { message(201);  goto uc2; }

  goto uc1;

uc2:  /* exit point */

  return(curc);
}


/****************************************************************************
 *                                                                          *
 *  homein -- from trial du to actual du for required s                     *
 *                                                                          *
 ****************************************************************************/

double
homein(p,j,k,u,du)
struct point *p ;
double u, du ;
int j, k ;
{ int i, loop = 0 ;
  double c, s, du0, du1 = 0.0, dx[NUMVAR], evalu() ;

  du0 = du ;

hm1:  /* binary search for required du */
```

```
  s = 0.0 ;
  param[j]->valu = u + du ;
  for (i = 0 ;  i < nvar ; i++)
  {
    dx[i] = (var[i]->valu = evalu(var[i]->f)) - p->x[i] ;
    s += dx[i] * dx[i] ;
  }
  s = sqrt(s) ;

#if VER == LATTICE  /* Lattice C version */
  if ( abs(s - (*dom[j])[k].trace) <= (*dom[j])[k].precision) goto hmx ;
  c =  abs(du1 - du) * 0.5 ;
#endif
#if VER == UNIX     /* Unix version      */
  if (fabs(s - (*dom[j])[k].trace) <= (*dom[j])[k].precision) goto hmx ;
  c = fabs(du1 - du) * 0.5 ;
#endif

  if (loop < LOOPMAX) { loop++; }
  else                { p->pflag = ERRLOOP;  du = 0.0;  goto hmx; }

  if (s > (*dom[j])[k].trace) { du1 = du;  du -= c;   } /* pulling upperbound */
  else if (du < du1)          {            du += c;   } /* upperbound is du1  */
  else                        {            du += du0; } /* pushing upperbound */

  if (du <= 0.0)      { p->pflag = ERRINCR;  du = 0.0;  goto hmx; }

  goto hm1;

hmx:  /* exit point */

  return(du);
}
```

```
/**************************************************************************
 *                                                                        *
 *  export.c -- functions used for exporting                              *
 *                                                                        *
 *  Author: Henri Cheung at McMaster University, Hamilton, Ontario        *
 *                                                                        *
 **************************************************************************/


#include  <stdio.h>
#include  "global.h"
#include  "struct.h"
#include  "struct2.h"
#include  "export.h"

extern FILE   *logf  ;
extern int    optlog ;
extern int    optmesh;
extern int    optaxis;
extern int    nparam, nvar ;
extern struct identifier *var[] ;
extern struct domain (*dom[])[NUMDOM] ;

extern struct curve   *cur2d ;
extern struct curve   *cur3d ;
extern struct surface *sur3d ;

FILE  *fout, *fw  ;
char  vid[IDWIDTH+1],  wid[IDWIDTH+1],  fid[IDWIDTH+1];
char  vhi0[IDWIDTH+1], vhi1[IDWIDTH+1], vhi[IDWIDTH+1];
char  vlo0[IDWIDTH+1], vlo1[IDWIDTH+1], vlo[IDWIDTH+1];


/**************************************************************************
 *                                                                        *
 *  export -- lead module of export shell                                 *
 *                                                                        *
 **************************************************************************/

export(filename)
char *filename;
{ struct curve   *c ;
  struct surface *s ;
  char cmd[STRSIZ], *work, *malloc(), *mktemp() ;

  /* open files */

  if ((work = malloc(STRSIZ)) == NULL)  message(504);

#if VER == LATTICE
  strcpy(work,"workfile");
#endif
#if VER == UNIX
  strcpy(work,"/tmp/tmpXXXXXX");
  work = mktemp(work);
#endif
```

```
  if ((fw = fopen(work,"w")) == NULL)  message(500) ;

  fout = (strcmp(filename,"stdout") == 0) ? stdout : fopen(filename,"w") ;
  if (fout == NULL)  message(503);

  /* initialize all counters */

  strncpy(vid,IDZERO,IDWIDTH);  vid[0] = 'V';  vid[IDWIDTH] = '\0';
  strncpy(wid,IDZERO,IDWIDTH);  wid[0] = 'W';  wid[IDWIDTH] = '\0';
  strncpy(fid,IDZERO,IDWIDTH);  fid[0] = 'F';  fid[IDWIDTH] = '\0';

  /* put colour map */

  putcmap();

  /* put axis min and max */

  if (optaxis == ON)  putlimit();

  /* put vertices, wires and surfaces */

  if (nparam == 1  &&  nvar == 2)
  {
    for (c = cur2d; c != NULL; c = c->nc)
    putcurve(c->i1,c->i2,c->nump,c->head);
  }
  else if (nparam == 1  &&  nvar == 3)
  {
    for (c = cur3d; c != NULL; c = c->nc)
    putcurve(c->i1,c->i2,c->nump,c->head);
  }
  else if (nparam == 2  &&  nvar == 3)
  {
    for (s = sur3d; s != NULL; s = s->ns)
    putsurface(s->i1,s->i2,s->numc,s->root);
  }

  /* define axis */

  if (optaxis == ON)  putaxis();

  /* append workfile to outfile, then remove[delete] workfile */

  fclose(fw) ;
  fflush(fout) ;
  if (strcmp(filename,"stdout") != 0)  fclose(fout) ;

#if VER == LATTICE  /*  Lattice C version  */
  strcpy(cmd,"type ");
#endif
#if VER == UNIX     /*  Unix version       */
  strcpy(cmd,"cat  ");
#endif
  strcat(cmd,work);
  if (strcmp(filename,"stdout") != 0)
  {
    strcat(cmd," >> ");    strcat(cmd,filename);
  }
  strcat(cmd,"\n");
  system(cmd);

  if (optlog == ON)  { fprintf(logf,"%s\n",cmd);  fflush(logf); }

#if VER == LATTICE  /*  Lattice C version  */
  strcpy(cmd,"del ");
#endif
#if VER == UNIX     /*  Unix version       */
  strcpy(cmd,"rm  ");
#endif
  strcat(cmd,work);  strcat(cmd,"\n");
  system(cmd);

  if (optlog == ON)  { fprintf(logf,"%s\n",cmd);  fflush(logf); }

}


/*****************************************************************************
 *                                                                           *
 *  putsurface -- put sur3d into UNIGRAFIX format                            *
 *                                                                           *
 *****************************************************************************/

putsurface(k0,k1,z,clo)
int k0, k1;
double z;
struct curve *clo;
{ struct curve *chi;
  struct point *phi, *plo;
  int ugrid, vgrid, ucount, vcount;
  int lofini = FALSE, hifini = FALSE;

  /* set grid size of current patch */

  ucount = ugrid = (int) ((*dom[0])[k0].display / (*dom[0])[k0].trace);
  vcount = vgrid = (int) ((*dom[1])[k1].display / (*dom[1])[k1].trace);

  if (clo == NULL)       goto es5;  /* no curve at all      */
  if ((int) z <= vgrid)  goto es5;  /* not enough grid size */

  /* dump vertex of low curve to file 1 */

  strcpy(vlo0,vid);  /* remember where vlo begins */

  if (((int) clo->nump) >= ugrid)
  for (plo = clo->head;  plo != NULL;  plo = (plo->pflag > 0) ? NULL : plo->np)
    if (ucount == ugrid || plo->np == NULL)
    {
      ucount = 1;  nextid(vid);
      fprintf(fout,"v %s %f %f %f ; {%f %f}\n",
              vid,plo->x[0],plo->x[1],plo->x[2],plo->u[0],plo->u[1]);
    }
    else ++ucount;

  strcpy(vlo1,vid);  /* remember where vlo stops */

  chi = clo;

es1:  /* next hi curve */

  if (chi->nc == NULL)  goto es4;

  while (vcount < vgrid && chi->nc != NULL)
  {
    ++vcount;  chi = chi->nc;
  }
  vcount = 0;

  /* dump vertex of high curve to file 1 */

  strcpy(vhi0,vid);  /* remember where vhi starts */

  if (((int) chi->nump) >= ugrid)
  for (ucount=ugrid, phi=chi->head; phi!=NULL; phi=(phi->pflag>0)?NULL:phi->np)
    if (ucount == ugrid || phi->np == NULL)
    {
      ucount = 1;  nextid(vid);
      fprintf(fout,"v %s %f %f %f ; {%f %f}\n",
              vid,phi->x[0],phi->x[1],phi->x[2],phi->u[0],phi->u[1]);
    }
    else ++ucount;

  strcpy(vhi1,vid);  /* remember where vhi stops */

  /* if either one is empty, get next set of curves */
  /* if v0 = v1, no point at all, skip              */
  /* otherwise process                              */

  if (strncmp(vlo0,vlo1,IDWIDTH) == 0) goto es3;
  if (strncmp(vhi0,vhi1,IDWIDTH) == 0) goto es3;

  /* set vertex counters */

  strcpy(vlo,vlo0);  nextid(vlo);
  strcpy(vhi,vhi0);  nextid(vhi);

es2:  /* link vertex as face(s) to file 2  */

  if (optmesh == OFF)
  {
    if ((lofini = (strncmp(vlo,vlo1,IDWIDTH) < 0) ? FALSE : TRUE) != TRUE)
    {
      nextid(fid);  fprintf(fw,"f %s (%s %s ",fid,vlo,vhi);
      nextid(vlo);  fprintf(fw,"%s);\n",vlo);
    }
    if ((hifini = (strncmp(vhi,vhi1,IDWIDTH) < 0) ? FALSE : TRUE) != TRUE)
    {
      nextid(fid);  fprintf(fw,"f %s (%s ",fid,vhi);
      nextid(vhi);  fprintf(fw,"%s %s);\n",vhi,vlo);
    }
  }
  else  /* mesh display */
  {
    nextid(wid);  fprintf(fw,"w %s (%s %s) RED ;\n",wid,vlo,vhi);
    if ((hifini = (strncmp(vhi,vhi1,IDWIDTH) < 0) ? FALSE : TRUE) != TRUE)
    {
      nextid(vhi);
    }
    if ((lofini = (strncmp(vlo,vlo1,IDWIDTH) < 0) ? FALSE : TRUE) != TRUE)
    {
      nextid(wid);  fprintf(fw,"w %s (%s ",wid,vlo);
      nextid(vlo);  fprintf(fw,"%s);\n",vlo);
    }
  }

  if (lofini != TRUE || hifini != TRUE) goto es2;

  /* last vertical segment */

  if (optmesh == ON)
  {
    nextid(wid);  fprintf(fw,"w %s (%s %s) RED ;\n",wid,vlo,vhi);
  }

es3:  /* next set of curves */

  strcpy(vlo0,vhi0);
  strcpy(vlo1,vhi1);
  clo = chi;
  goto es1;

es4:  /* last curve */

  if (optmesh == ON)
  {
    strcpy(vlo,vlo0);  nextid(vlo);
    while (strncmp(vlo,vlo1,IDWIDTH) < 0)
    {
      nextid(wid);  fprintf(fw,"w %s (%s ",wid,vlo);
      nextid(vlo);  fprintf(fw,"%s);\n",vlo);
    }
  }

es5:  /* exit point */
  ;
}


/*****************************************************************************
 *                                                                           *
 *  putcurve - put cur2d / cur3d into UNIGRAFIX format                       *
 *                                                                           *
 *****************************************************************************/
```

```
putcurve(j,k,x,p)
int j, k;
double x;
struct point *p;
{ int n, nseg = -1;
  int counter, roof ;

  counter = roof = (int)((*dom[j])[k].display / (*dom[j])[k].trace) ;

ec1:

  if (p == NULL) goto ec2 ;

  if (counter < roof)
  {
    p = p->np;  ++counter;  goto ec1;
  }
  else counter = 1;

  /* dump point as vertex to file 1 */

  nextid(vid);
  fprintf(fout,"v %s %f %f %f ; {%f}\n",vid,p->x[0],p->x[1],p->x[2],p->u[0]);
  p = p->np;

  /* link vertex as wire to file 2  */

  if (nseg == -1)  /* first segment */
  {
    nseg = 0;  nextid(wid);  fprintf(fw,"w %s (%s ",wid,vid);
  }
  else if (nseg < WSEGLEN)
  {
    nseg++;  fprintf(fw,"%s ",vid);
  }
  else  /* start a new segment */
  {
    nseg = 0;  fprintf(fw,"%s);\n",vid);
    nextid(wid);  fprintf(fw,"w %s (%s ",wid,vid);
  }

  if (p->pflag > 0)
  switch (p->pflag)
  {
    case 1: message(751);  break;
    case 2: message(752);  break;
    case 3: message(504);  break;
  }

  goto ec1;

ec2: /* wrap up procedures */

  if (nseg == 0)  fprintf(fw,"%s ",vid);
  fprintf(fw,");\n");
```

```
}


/***************************************************************************
 *                                                                         *
 *  nextid -- increment id counter                                         *
 *                                                                         *
 ***************************************************************************/

nextid(ic)
char ic[];
{ int d, carry = ON ;

  d = IDWIDTH - 1 ;

  while (carry == ON  &&  d > 0)
  switch(ic[d])
  {
    case '0': ic[d--] = '1' ;  carry = OFF ;  break ;
    case '1': ic[d--] = '2' ;  carry = OFF ;  break ;
    case '2': ic[d--] = '3' ;  carry = OFF ;  break ;
    case '3': ic[d--] = '4' ;  carry = OFF ;  break ;
    case '4': ic[d--] = '5' ;  carry = OFF ;  break ;
    case '5': ic[d--] = '6' ;  carry = OFF ;  break ;
    case '6': ic[d--] = '7' ;  carry = OFF ;  break ;
    case '7': ic[d--] = '8' ;  carry = OFF ;  break ;
    case '8': ic[d--] = '9' ;  carry = OFF ;  break ;
    case '9': ic[d--] = 'A' ;  carry = OFF ;  break ;
    case 'A': ic[d--] = 'B' ;  carry = OFF ;  break ;
    case 'B': ic[d--] = 'C' ;  carry = OFF ;  break ;
    case 'C': ic[d--] = 'D' ;  carry = OFF ;  break ;
    case 'D': ic[d--] = 'E' ;  carry = OFF ;  break ;
    case 'E': ic[d--] = 'F' ;  carry = OFF ;  break ;
    case 'F': ic[d--] = '0' ;  carry =  ON ;  break ;
  }

  if (strncmp(ic,IDZERO,IDWIDTH) == 0)  message(702);
}


/***************************************************************************
 *                                                                         *
 *  putlimit -- put axis min & max                                         *
 *                                                                         *
 ***************************************************************************/

putlimit()
{ int i, j;
  double z;

  for (i = 0;  i < nvar;  i++)
  {
    /* minimum point */

    fprintf(fout,"v x%ldmin  ",i+1);
    for (j = 0;  j < i;  j++)
    fprintf(fout,"0.0  ");
    fprintf(fout,"%f  ",var[i]->vmin);
    for (j++; j < VARMAX; j++)
    fprintf(fout,"0.0  ");
    fprintf(fout,"; \n");

    /* maximum point */

    fprintf(fout,"v x%ldmax  ",i+1);
    for (j = 0;  j < i;  j++)
    fprintf(fout,"0.0  ");
    fprintf(fout,"%f  ",var[i]->vmax);
    for (j++; j < VARMAX; j++)
    fprintf(fout,"0.0  ");
    fprintf(fout,"; \n");

    /* axis marker   */

    z = var[i]->vmax + 0.05 * (var[i]->vmax - var[i]->vmin);

    fprintf(fout,"v x%ldplus ",i+1);
    for (j = 0;  j < i;  j++)
    fprintf(fout,"0.0  ");
    fprintf(fout,"%f  ",z);
    for (j++; j < VARMAX; j++)
    fprintf(fout,"0.0  ");
    fprintf(fout,"; \n");
  }
}


/***************************************************************************
 *                                                                         *
 *  putaxis -- put axis                                                    *
 *                                                                         *
 ***************************************************************************/

putaxis()
{ int i;

  for (i = 0;  i < nvar;  i++)
  fprintf(fout,"w x%ldaxis (x%ldmin x%ldmax) GREEN; \n",i+1,i+1,i+1);

  fprintf(fout,"w x1mark (x1max x1plus) RED    ; \n");
  fprintf(fout,"w x2mark (x2max x2plus) YELLOW ; \n");
  if (nvar == 3)
  fprintf(fout,"w x3mark (x3max x3plus) BLACK  ; \n");
}


/***************************************************************************
 *                                                                         *
 *  putcmap -- put colour map                                              *
 *                                                                         *
 ***************************************************************************/
```

```
putcmap()
{
  fprintf(fout,"c  BLACK     0.0    0   0 ; \n");
  fprintf(fout,"c  RED       1.0    0   1 ; \n");
  fprintf(fout,"c  YELLOW    1.0   60   1 ; \n");
  fprintf(fout,"c  GREEN     1.0  120   1 ; \n");
  fprintf(fout,"c  CYAN      1.0  180   1 ; \n");
  fprintf(fout,"c  BLUE      1.0  240   1 ; \n");
  fprintf(fout,"c  WHITE     1.0  360   0 ; \n");
}
```

```
/*************************************************************************
 *                                                                       *
 *  message.c -- message file                                            *
 *                                                                       *
 *  Author: Henri Cheung at McMaster University, Hamilton, Ontario       *
 *                                                                       *
 *************************************************************************/


#include  <stdio.h>
#include  "global.h"
#include  "message.h"

extern int   optlog;
extern FILE *logf  ;
extern char *logID ;

message (msgcode)
int msgcode ;
{
  /* print appropriate message to screen */

  switch (msgcode)
  {
    case   1: fprintf(stderr,MSG001);   break;
    case   2: fprintf(stderr,MSG002);   break;
    case   3: fprintf(stderr,MSG003);   break;
    case   4: fprintf(stderr,MSG004);   break;
    case   5: fprintf(stderr,MSG005);   break;
    case   6: fprintf(stderr,MSG006);   break;
    case   7: fprintf(stderr,MSG007);   break;
    case   8: fprintf(stderr,MSG008);   break;
    case   9: fprintf(stderr,MSG009);   break;
    case  10: fprintf(stderr,MSG010);   break;
    case 500: fprintf(stderr,MSG500);   break;
    case 501: fprintf(stderr,MSG501);   break;
    case 502: fprintf(stderr,MSG502);   break;
    case 503: fprintf(stderr,MSG503);   break;
    case 504: fprintf(stderr,MSG504);   break;
    case 505: fprintf(stderr,MSG505);   break;
    case 506: fprintf(stderr,MSG506);   break;
    case 507: fprintf(stderr,MSG507);   break;
    case 508: fprintf(stderr,MSG508);   break;
    case 509: fprintf(stderr,MSG509);   break;
    case 510: fprintf(stderr,MSG510);   break;

    case 511: fprintf(stderr,MSG511);   break;
    case 512: fprintf(stderr,MSG512);   break;
    case 513: fprintf(stderr,MSG513);   break;
    case 514: fprintf(stderr,MSG514);   break;
    case 515: fprintf(stderr,MSG515);   break;
    case 516: fprintf(stderr,MSG516);   break;
    case 517: fprintf(stderr,MSG517);   break;

    case 521: fprintf(stderr,MSG521);   break;
    case 522: fprintf(stderr,MSG522);   break;
case 523: fprintf(stderr,MSG523);   break;
case 524: fprintf(stderr,MSG524);   break;
case 525: fprintf(stderr,MSG525);   break;
case 526: fprintf(stderr,MSG526);   break;
case 527: fprintf(stderr,MSG527);   break;

case 531: fprintf(stderr,MSG531);   break;
case 532: fprintf(stderr,MSG532);   break;
case 533: fprintf(stderr,MSG533);   break;
case 534: fprintf(stderr,MSG534);   break;
case 535: fprintf(stderr,MSG535);   break;
case 536: fprintf(stderr,MSG536);   break;

case 541: fprintf(stderr,MSG541);   break;
case 542: fprintf(stderr,MSG542);   break;
case 543: fprintf(stderr,MSG543);   break;
case 544: fprintf(stderr,MSG544);   break;
case 545: fprintf(stderr,MSG545);   break;

case 551: fprintf(stderr,MSG551);   break;
case 552: fprintf(stderr,MSG552);   break;
case 553: fprintf(stderr,MSG553);   break;
case 554: fprintf(stderr,MSG554);   break;
case 555: fprintf(stderr,MSG555);   break;
case 556: fprintf(stderr,MSG556);   break;
case 557: fprintf(stderr,MSG557);   break;

case 561: fprintf(stderr,MSG561);   break;
case 571: fprintf(stderr,MSG571);   break;

case 581: fprintf(stderr,MSG581);   break;
case 582: fprintf(stderr,MSG582);   break;
case 583: fprintf(stderr,MSG583);   break;

case 591: fprintf(stderr,MSG591);   break;
case 592: fprintf(stderr,MSG592);   break;

case 101: fprintf(stderr,MSG101);   break;
case 102: fprintf(stderr,MSG102);   break;
case 103: fprintf(stderr,MSG103);   break;
case 104: fprintf(stderr,MSG104);   break;
case 105: fprintf(stderr,MSG105);   break;
case 106: fprintf(stderr,MSG106);   break;
case 107: fprintf(stderr,MSG107);   break;
case 108: fprintf(stderr,MSG108);   break;
case 601: fprintf(stderr,MSG601);   break;
case 602: fprintf(stderr,MSG602);   break;
case 603: fprintf(stderr,MSG603);   break;
case 604: fprintf(stderr,MSG604);   break;
case 605: fprintf(stderr,MSG605);   break;
case 606: fprintf(stderr,MSG606);   break;
case 607: fprintf(stderr,MSG607);   break;

case 201: fprintf(stderr,MSG201);   break;

case 251: fprintf(stderr,MSG251);   break;
```

```
    case 252: fprintf(stderr,MSG252);   break;

    case 701: fprintf(stderr,MSG701);   break;
    case 702: fprintf(stderr,MSG702);   break;
    case 703: fprintf(stderr,MSG703);   break;

    case 751: fprintf(stderr,MSG751);   break;
    case 752: fprintf(stderr,MSG752);   break;

    default:  fprintf(stderr,MSGYYY);   break;
  }

  /* print appropriate message to logfile */

  if (optlog == ON)
  switch (msgcode)
  {
    case   1: fprintf(logf,MSG001);     break;
    case   2: fprintf(logf,MSG002);     break;
    case   3: fprintf(logf,MSG003);     break;
    case   4: fprintf(logf,MSG004);     break;
    case   5: fprintf(logf,MSG005);     break;
    case   6: fprintf(logf,MSG006);     break;
    case   7: fprintf(logf,MSG007);     break;
    case   8: fprintf(logf,MSG008);     break;
    case   9: fprintf(logf,MSG009);     break;
    case  10: fprintf(logf,MSG010);     break;
    case 500: fprintf(logf,MSG500);     break;
    case 501: fprintf(logf,MSG501);     break;
    case 502: fprintf(logf,MSG502);     break;
    case 503: fprintf(logf,MSG503);     break;
    case 504: fprintf(logf,MSG504);     break;
    case 505: fprintf(logf,MSG505);     break;
    case 506: fprintf(logf,MSG506);     break;
    case 507: fprintf(logf,MSG507);     break;
    case 508: fprintf(logf,MSG508);     break;
    case 509: fprintf(logf,MSG509);     break;
    case 510: fprintf(logf,MSG510);     break;

    case 511: fprintf(logf,MSG511);     break;
    case 512: fprintf(logf,MSG512);     break;
    case 513: fprintf(logf,MSG513);     break;
    case 514: fprintf(logf,MSG514);     break;
    case 515: fprintf(logf,MSG515);     break;
    case 516: fprintf(logf,MSG516);     break;
    case 517: fprintf(logf,MSG517);     break;

    case 521: fprintf(logf,MSG521);     break;
    case 522: fprintf(logf,MSG522);     break;
    case 523: fprintf(logf,MSG523);     break;
    case 524: fprintf(logf,MSG524);     break;
    case 525: fprintf(logf,MSG525);     break;
    case 526: fprintf(logf,MSG526);     break;
    case 527: fprintf(logf,MSG527);     break;

    case 531: fprintf(logf,MSG531);     break;
```

```
case 532: fprintf(logf,MSG532);      break;
case 533: fprintf(logf,MSG533);      break;
case 534: fprintf(logf,MSG534);      break;
case 535: fprintf(logf,MSG535);      break;
case 536: fprintf(logf,MSG536);      break;

case 541: fprintf(logf,MSG541);      break;
case 542: fprintf(logf,MSG542);      break;
case 543: fprintf(logf,MSG543);      break;
case 544: fprintf(logf,MSG544);      break;
case 545: fprintf(logf,MSG545);      break;

case 551: fprintf(logf,MSG551);      break;
case 552: fprintf(logf,MSG552);      break;
case 553: fprintf(logf,MSG553);      break;
case 554: fprintf(logf,MSG554);      break;
case 555: fprintf(logf,MSG555);      break;
case 556: fprintf(logf,MSG556);      break;
case 557: fprintf(logf,MSG557);      break;

case 561: fprintf(logf,MSG561);      break;
case 571: fprintf(logf,MSG571);      break;

case 581: fprintf(logf,MSG581);      break;
case 582: fprintf(logf,MSG582);      break;
case 583: fprintf(logf,MSG583);      break;

case 591: fprintf(logf,MSG591);      break;
case 592: fprintf(logf,MSG592);      break;

case 101: fprintf(logf,MSG101);      break;
case 102: fprintf(logf,MSG102);      break;
case 103: fprintf(logf,MSG103);      break;
case 104: fprintf(logf,MSG104);      break;
case 105: fprintf(logf,MSG105);      break;
case 106: fprintf(logf,MSG106);      break;
case 107: fprintf(logf,MSG107);      break;
case 108: fprintf(logf,MSG108);      break;
case 601: fprintf(logf,MSG601);      break;
case 602: fprintf(logf,MSG602);      break;
case 603: fprintf(logf,MSG603);      break;
case 604: fprintf(logf,MSG604);      break;
case 605: fprintf(logf,MSG605);      break;
case 606: fprintf(logf,MSG606);      break;
case 607: fprintf(logf,MSG607);      break;

case 201: fprintf(logf,MSG201);      break;

case 251: fprintf(logf,MSG251);      break;
case 252: fprintf(logf,MSG252);      break;

case 701: fprintf(logf,MSG701);      break;
case 702: fprintf(logf,MSG702);      break;
case 703: fprintf(logf,MSG703);      break;

case 751: fprintf(logf,MSG751);      break;
```

```
    case 752: fprintf(logf,MSG752);      break;

    default:  fprintf(logf,MSGYYY);      break;
  }
  if (optlog == ON)  fflush(logf);

  /* abort program if error is fatal */

  if (msgcode > 499)
  {
    fprintf(stderr,MSGXXX);
    if (optlog == ON)
    {
      fprintf(stderr,"\n[Logfile is %s]\n",logID);
      fprintf(logf,"%s\n[Logfile is %s]\n",MSGXXX,logID);  fclose(logf);
    }
    exit(1);    /* opened files are closed automatically */
  }
}
```

```
/**************************************************************************
 *                                                                        *
 *  audit.c -- print expression tree to screen for error checking         *
 *                                                                        *
 *  Author: Henri Cheung at McMaster University, Hamilton, Ontario        *
 *                                                                        *
 **************************************************************************/


#include  <stdio.h>
#include  "global.h"
#include  "struct.h"
#include  "struct2.h"

extern FILE *logf ;
extern int  optlog;
extern int  nconst, nvar;
extern struct identifier *const[], *var[];


/**************************************************************************
 *                                                                        *
 *  audit -- prints all expression from its tree form                     *
 *                                                                        *
 **************************************************************************/

audit ()
{ int i;

  if (optlog == ON)
  {
    for (i = 0; i < nconst; i++)
    {
      fprintf(logf,"%s = ",const[i]->nam);
      prtexpr(const[i]->f);
      fprintf(logf," = %f \n",const[i]->valu);
    }
    for (i = 0; i < nvar; i++)
    {
      fprintf(logf,"%s = ",var[i]->nam);
      prtexpr(var[i]->f);
      fprintf(logf,"\n");
    }
    fflush(logf);
  }
}


/**************************************************************************
 *                                                                        *
 *  prtexpr -- recursive print shell                                      *
 *                                                                        *
 **************************************************************************/

prtexpr(f)
struct expr *f;
```

```
{
  if (f != NULL)
  switch(f->ctype)
  {
    case CONSTID :
    case PARAMID :
    case VARID    : fprintf(logf,"%s",f->info.ptr->nam); break;
    case INTEGER : fprintf(logf,"%d",f->info.ival);     break;
    case REAL     : fprintf(logf,"%f",f->info.rval);     break;
    case OPERATOR: prtop(f);                             break;
    case FUNCTION: fprintf(logf,"%s(",f->info.sval);
                       prtexpr(f->r); fputc(')',logf);  break;
    default:        fprintf(logf,"?");                   break;
  }
}


/***********************************************************************
 *                                                                     *
 *  prtop -- print a operator node                                     *
 *                                                                     *
 ***********************************************************************/

prtop(f)
struct expr *f;
{
  /* process left child of current node */

  if (f->l->ctype != OPERATOR)
  {
    prtexpr(f->l);
  }
  else if (*(f->info.sval) == '^')
  {
    fputc('(',logf);  prtexpr(f->l);  fputc(')',logf);
  }
  else if ( ((*(f->info.sval) == '*') || (*(f->info.sval) == '/')) &&
            ((*(f->l->info.sval) == '+') || (*(f->l->info.sval) == '-')))
  {
    fputc('(',logf);  prtexpr(f->l);  fputc(')',logf);
  }
  else prtexpr(f->l);

  /* print operator */

  if (*(f->info.sval) != '^')  fputc(' ',logf);
  fputc(*(f->info.sval),logf);
  if (*(f->info.sval) != '^')  fputc(' ',logf);

  /* process right child */

  if (f->r != NULL)
  if (f->r->ctype != OPERATOR)
  {
    prtexpr(f->r);
  }
  else
  {
    fputc('(',logf);  prtexpr(f->r);  fputc(')',logf);
  }
}


/************************************************************************
 *                                                                      *
 *  logsurface -- log surface to logfile                                *
 *                                                                      *
 ************************************************************************/

logsurface(s)
struct surface *s;
{ struct curve *c;

  fprintf(logf,
        "Surface has %12.1f curve(s), %12.1f point(s)\n",s->numc,s->nump);
  for (c = s->root;  c != NULL;  c = c->nc)  logcurve(c,3);
  fflush(logf);
}


/************************************************************************
 *                                                                      *
 *  logcurve -- log curve to logfile                                    *
 *                                                                      *
 ************************************************************************/

logcurve(c,d)
int d;
struct curve *c;
{ struct point *p;

  fprintf(logf,"\tCurve has %12.1f point(s)\n",c->nump);
  for (p = c->head;  p != NULL;  p = p->np)  logpoint(p,d);
  fflush(logf);
}


/************************************************************************
 *                                                                      *
 *  logpoint -- log point to logfile                                    *
 *                                                                      *
 ************************************************************************/

logpoint(p,d)
int d;
struct point *p;
{
  switch (d)
  {
    case 2: fprintf(logf,"%12.5f | %12.5f %12.5f\n",p->u[0],p->x[0],p->x[1]);
            break;
    case 3: fprintf(logf,"%12.5f %12.5f | %12.5f %12.5f %12.5f\n",
                                        p->u[0],p->u[1],p->x[0],p->x[1],p->x[2]);
           break;
    default:break;
  }
  switch (p->pflag)
  {
    case ERRLOOP: message(251);  break;
    case ERRINCR: message(252);  break;
    case ERRMEM:  message(504);  break;
    default:break;
  }
}
```

```
/****************************************************************
 *                                                              *
 *  wtty.c -- updates file .server_tty at user's home directory *
 *                                                              *
 *   user's .login file should contain a line: setenv TTY `tty` *
 *   user's .cshrc file should contain a line:                  *
 *    alias grterm 'setenv GRTERM `cat ~user_name/.server_tty` ' *
 *                                                              *
 *  notes:                                                      *
 *                                                              *
 *  - /etc/utmp is a system file which contains information of  *
 *    who's currently login & their terminal file address       *
 *    (utmp(5)).  The file is a sequence of record of           *
 *    definition:                                               *
 *                                                              *
 *              struct utmp                                     *
 *              {                                               *
 *                char ut_line[8];                              *
 *                char ut_name[8];                              *
 *                char ut_host[16];                             *
 *                long ut_time;                                 *
 *              }                                               *
 *                                                              *
 *   However, "/dev" is not included in the field ut_line.      *
 *   In other words, the terminal address should be "/dev"      *
 *   + ut_line                                                  *
 *                                                              *
 * - by obtaining user name from the environment variable USER, *
 *   & another preset environment variable TTY, any server      *
 *   (remote) terminal, if exist can be identified & stored in  *
 *    some data files, e.g. .server_tty                         *
 *                                                              *
 * - possible further expansion: implement an array of tty's;   *
 *   user's choice                                              *
 *                                                              *
 *   Author:                                                    *
 *   Henri Cheung at McMaster University, Hamilton, Ontario     *
 *                                                              *
 ****************************************************************/


#include  <stdio.h>
#include  <utmp.h>

#define  STRLEN 32
#define  UTMP "/etc/utmp"  /* system file
                              which contains login info */

extern char **environ;


main()
{ int  i, fd, found = 0;
  char fname[STRLEN], user_name[STRLEN];
  char current_tty[STRLEN], other_tty[STRLEN];
  struct utmp buf;
  /* copy environment variable USER */

  strcpy(user_name,getenv("USER"));
  strcpy(current_tty,getenv("TTY"));

  /* get first tty address under user name but not current tty */

  fd = open(UTMP,0);
  while (!found  &&  read(fd,&buf,sizeof(buf)) > 0)
  if (strcmp(user_name,buf.ut_name) == 0)
  {
    sprintf(other_tty,"/dev/%s",buf.ut_line);
    if (strcmp(current_tty,other_tty) != 0)  found = 1;
  }
  close(fd);

  /* update ~user_name/.server_tty */

  if (found)
  {
    sprintf(fname,"%s/.server_tty",getenv("HOME"));
    fd = creat(fname,0644);
    write(fd,other_tty,sizeof(other_tty));
    close(fd);
  }

  /* report status */

  printf("\n");
  printf("Current tty address: %s (%s)\n",current_tty,user_name);
  if (found)
  {
    printf("            also at: %s\n",other_tty);
    printf("To set GRTERM, type grterm<cr> \n");
  }
  printf("\n");

  for (i=0; *environ[i] != '\0'; i++)  printf("%s\n",environ[i]);
}
```