A USER-FRIENDLY INTERFACE FOR A POISSON-SOLVER

by

Ted C. Johnson

SUBMITTED TO THE DEPARTMENT OF
ELECTRICAL ENGINEERING AND COMPUTER
SCIENCE IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE OF

BACHELOR OF SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 1987

Signature of Author Signature redacted
Department of Electrical Engineering and Computer Science
June 1, 1987

Certified by ______________________
Professor Abraham Bers
Thesis Supervisor

Accepted by ______________________
Professor David Adler
Chairman, Undergraduate Thesis Committee

Signature redacted

A USER-FRIENDLY INTERFACE FOR A POISSON-SOLVER

by

Ted C. Johnson

Submitted to the Department of Electrical Engineering and Computer Science on June 1, 1987 in partial fulfillment of the requirements for the degree of Bachelor of Science.

Abstract

This thesis describes a new interface for an existing Poisson-solver, and enhancements to the displaying of the results. The new interface is menu-driven, and uses a graphics editor to greatly simplify the task of entering problems of interest via a computer keyboard.

Dedication

I would like to thank my parents, Eva Vanderbilt and Essex Johnson, for their love and support over the years... I'm almost done now!

I would also like to thank Prof. Bers, for his help and guidance, and my girlfriend Cindy Roberts, for putting up with my obnoxiousness.

Table of Contents

List of Figures

Chapter 1

Introduction

1.1 Introduction.

Poisson's equation is used in the study of static electric fields. In electrostatic (ES) problems, we are only concerned with the electric field, $\bar{E}$. The two Maxwell equations which govern the behavior of ES electric fields are:

Gauss' electric law: $\nabla \cdot \epsilon_0 \bar{E} = \rho$ (1-1)

Faraday's law: $\nabla \times \bar{E} = 0$ (1-2)

Any vector function whose curl is zero can be expressed as the gradient of a scalar function. Therefore Faraday's law implies that the electric field strength vector, $\bar{E}$, can be expressed as the gradient of a scalar function. By convention, the scalar function Φ (which is called the electric potential) is defined[1] such that:

$$\bar{E} = -\nabla\bar{\Phi} \qquad (1\text{-}3)$$

Substituting $-\nabla\bar{\Phi}$ in for $\bar{E}$ in Gauss' law, we obtain Poisson's equation:

$$\nabla^2\Phi = -\frac{\rho}{\epsilon_0} \qquad (1\text{-}4)$$

We have just replaced the vector function in the two vector differential Eqs. (1-1) and (1-2) with a scalar function that contains the same information. And because Eq. (1-4) is a combination of Eqs. (1-1) and (1-2), it contains all of the information that they do; thus Poisson's equation is (in electrostatic situations) "shorthand" for the two Maxwell equations which deal with the electric field. This is nice, but it's not the reason why Poisson's equation is so useful. Poisson's equation is useful because it replaces a vector function with a scalar function, which is much easier to work with.

---

1 By convention, electric field lines point from higher electric potential to lower electric potential. The gradient of the electric potential, $\nabla\bar{\Phi}$, points in the direction of the greatest increase in $\bar{\Phi}$, thus it points in the opposite direction of the electric field lines. The purpose of the minus sign in Eq. (1-3) is to indicate this fact.

1.2 How to state an ES problem.

An electrostatic problem is stated by specifying (1)the distribution of electric charge density, $\rho(x,y,z)$, within a volume of interest and (2)the boundary conditions, e.g., the distribution of electric potential, $\Phi(x,y,z)$ on the boundaries of the volume of interest.

1.3 How to solve an ES problem.

The solution to an ES problem is found by finding an electric potential function, $\Phi(x,y,z)$, which satisfies BOTH Poisson's equation and the boundary conditions. From this electric potential function, the electric field strength vector, $\bar{E}$, for the particular ES problem can be found via Eq. (1-3). To visualize the solution to an ES problem it is useful to graph the equipotential contours (i.e., the lines where $\Phi(x,y,z)$ = constant), and the associated $\bar{E}$ field lines.

1.4 What is a Poisson-solver?

A Poisson-solver is a computer program that numerically solves Poisson's equation inside of a given region of two-dimensional (x,y) space. It lets the user specify the distribution of charge and/or potentials inside the region of interest, and the boundary conditions on the region. It then solves for the potential everywhere inside the region. The resulting equipotential contours and the electric field lines are then displayed on the user's terminal.

In general, Poisson-solvers are useful because most problems cannot be solved analytically, and must be solved via numerical methods. A FAST Poisson-solver is useful because it allows a student to examine quickly a large variety of ES problems in a few minutes instead of several hours (which is how long it would take if the student solved all of the problems by hand). By doing the hard work for the student, a Poisson-solver encourages the student to explore all sorts of ES problems that he may have been curious about. This is useful, because the student can thus develop his sense of intuition for what the solutions to different ES problems might be like.

1.5 Description of "Poisson".

1.5.1 General description.

The Poisson-solver I worked on is called "Poisson" [see Appendix A]. It solves Poisson's equation in two dimensions, namely, (x,y) space, via the finite-difference relaxation method. This system was implemented by Denise Barnett in collaboration with Greg Francis and and Prof. Abraham Bers [see Appendix A].

"Poisson" is restricted to two-dimensional ES problems! Everything (i.e., the boundaries and the charges) is infinitely extended, uniform in the z-direction. Nothing varies in the z-direction [see Figure 1-1]. Therefore, we are solving the restricted 2-dimensional version of Poisson's equation:

$$\frac{\partial^2}{\partial x^2}\Phi(x,y) + \frac{\partial^2}{\partial y^2}\Phi(x,y) = \frac{-\rho(x,y)}{\epsilon_0} \qquad (1\text{-}5)$$

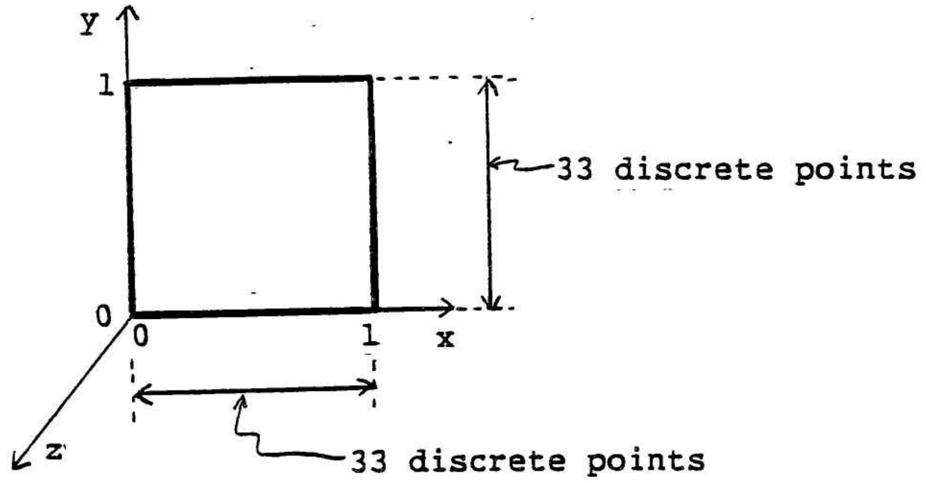


Figure 1-1: "Poisson"'s coordinate system.

The user enters the problem in a square region, called the "problem box", which is of unit length in the x-direction and in the y-direction. There are an infinite number of points between 0.0 and 1.0, however computers must deal with continuous distributions by approximating them as a series of discrete values; "Poisson" approximates the infinite number of points in the range 0.0 <= x <= 1.0, 0.0 <= y <= 1.0 by a grid of 33x33 points. The top and bottom rows and the left and right columns of this grid are occupied by the boundaries of the problem box, which means that there are 31x31 internal grid points. The bold-faced lines in Figure 1-1 are the four boundaries (top, bottom, left, and right) of the problem box.

The user can specify a potential, $\Phi(x,y) = f(x,y)$, or a line charge density, $\lambda(x,y)$ = constant, at any of the 31x31 internal grid points. The charge density at a point (x_0,y_0) is given by $\rho(x,y) = \lambda\delta(x-x_0)\delta(y-y_0)$. In order for the solution to be unique, the potential, xor the normal derivative of the potential, must be specified on each of the problem box's four boundaries. Such specification of the problem is known to lead to a unique solution.

1.5.2 Specifying the boundary conditions of an ES problem.

1.5.2.1 Defining the boundaries.

The boundaries are the four sets of 33 grid points which make up the "walls" of the "problem box" [see Figure 1-1]. It is along these four boundaries that the boundary conditions must be established. For each boundary, one of the following must be specified: either (1)the potential at each point, xor (2)the normal derivative of the potential at each point.

1.5.2.2 Types of boundary conditions "Poisson" can handle.

If all four boundaries have their potentials specified, then the problem box is said to have "Dirichlet boundary conditions". If all four boundaries have the normal derivatives of their potentials specified, then the problem box is said to have "Neumann boundary conditions". However, if the potential is specified on some of the boundaries, but the normal derivative of the potential is specified on the others, then the problem is said to have "mixed Dirichlet and Neumann boundary conditions". "Poisson" can handle all three cases (i.e., Dirichlet, Neumann, and mixed Dirichlet and Neumann boundary conditions).

1.5.2.3 Functions available for specifying the boundary conditions.

For each of the four boundaries, the user can specify one of five functions for the potential along that boundary (xor for the normal derivative of the potential along that boundary). The five functions are:

```
(1)constant ---- the user must specify the constant.

(2)linear ------ a ramp up from a given beginning
                 value to a given ending value.
                 The user must specify both values.

(3)step -------- half the boundary is value #1; the
                 other half is value #2.  The user
                 must specify both values.

(4)sine -------- a*sin(b*pi*x), where the user must
                 specify a and b.

(5)cosine ------ a*cos(b*pi*x), where the user must
                 specify a and b.
```

The suggested range of boundary potentials is $|v| <= 10.0$ volts.

1.5.2.4 Specifying the position and value of equipotentials (sheets and/or rods) in the interior of the problem region.

The user can specify the value of an equipotential rod or the value of an equipotential sheet at any of the 31x31 internal grid points. The valid range of voltages is |V| <= 10.0 volts [see Appendix A]. The voltage is limited to this range to speed up the Poisson-solver. If, for the same boundary potentials, a wider range of values is allowed for the equipotential rods or sheets, say -100.0 to +100.0 volts, the gradients in the problem are increased and it will take the Poisson-solver many more iterations (i.e., a much longer time) to converge to the correct solution.

Since the problem box is two-dimensional, all of the regions of constant potential are either rods or sheets.

1.5.3 Specifying the distribution of the sources (i.e., line charges) in the interior of the problem region.

The only sources available in "Poisson" are line charges. The valid range of values for lambda is 1.e-12 <= |lambda| <= 1.e-10 [see Appendix A]. Once again, the reason for limiting the range of lambda is to speed up the Poisson-solver.

All of the sources must be in the interior of the problem region, i.e., at one of the 31x31 grid points. A source cannot appear on or outside any of the four walls which bound the square problem region.

Chapter 2
The new interface.

The new interface is much more friendly than the old interface was. Refer to Appendix B for a description of the old interface. The very first thing that the new "Poisson" interface does is to greet the user with the following:

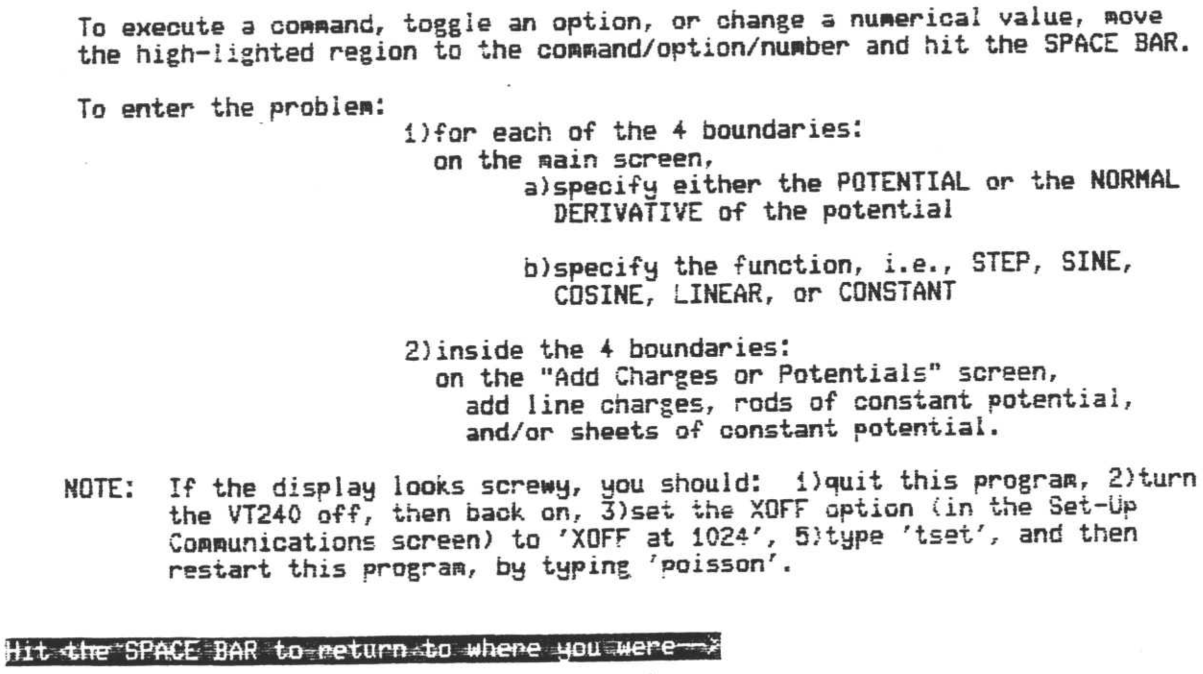
```
           A Solution to Poisson's Equation within a Square Region
             with Dirichlet and/or Neumann Boundary Conditions
                        Version 1.0  October 1986
     This program was intended to provide means for quickly (in 5-10 seconds)
solving relatively simple problems in electrostatics and/or magnetostatics.
     On the boundaries of the region a variety of functions of potentials
(Dirichlet) or normal derivatives of potential (Neumann) may be specified.
Inside the region, potential rods and/or sheets, as well as charged rods, can
be specified.  The solution is displayed as contour plots of the potentials.
The field lines are then displayed as contours of the conjugate potential.
     Problems are solved on a 33x33 point grid (32x32 spaces).  For a
"smoother" plot with more waiting time (factor of four), the grid size can
be changed to 65x65 points (64x64 spaces).
     In the interest of achieving the desired speed, some suggested ranges
for the values to be entered in any problem are listed below.  It is possible
to enter values outside of these ranges but computation time will increase.

                      1 <= | potentials | <= 10 Volts
     1e-2 <= | normal derivatives of potential | <= 1e-1 Volts/unit length
        2e-11 <= | line charge densities | <= 2e-10 Coulomb/unit length

     For further details on this program, as well as a demo, refer to the
writeup:  "Project Athena Poisson Program" for course 6.013.
     Press RETURN for main menu to specify a particular problem.
```

Figure 2-1: "Poisson"'s introductory message.

Many of the shortcomings of the old interface were resolved in a fairly straightforward manner. These improvements will be discussed first; the big leap in user-friendliness (i.e., the graphics editor) will be discussed last.

2.1 Miscellaneous improvements.

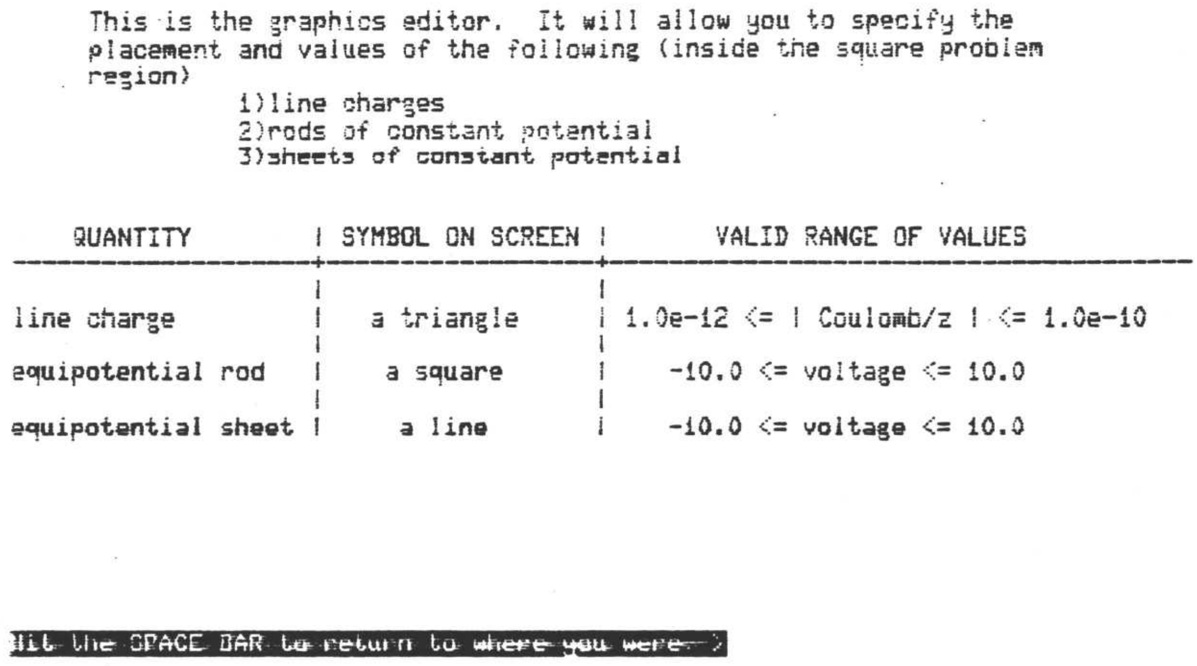
There is now a help facility. If the user is on the main screen and selects the "Help" menu item, the main screen will be erased and he will see:



```
  To execute a command, toggle an option, or change a numerical value, move
  the high-lighted region to the command/option/number and hit the SPACE BAR.

  To enter the problem:
                           1)for each of the 4 boundaries:
                             on the main screen,
                                   a)specify either the POTENTIAL or the NORMAL
                                     DERIVATIVE of the potential

                                   b)specify the function, i.e., STEP, SINE,
                                     COSINE, LINEAR, or CONSTANT

                           2)inside the 4 boundaries:
                             on the "Add Charges or Potentials" screen,
                               add line charges, rods of constant potential,
                               and/or sheets of constant potential.

 NOTE:  If the display looks screwy, you should:  1)quit this program, 2)turn
        the VT240 off, then back on, 3)set the XOFF option (in the Set-Up
        Communications screen) to 'XOFF at 1024', 5)type 'tset', and then
        restart this program, by typing 'poisson'.

Hit the SPACE BAR to return to where you were -->
```

<u>Figure 2-2</u>: "Help" message for main screen.

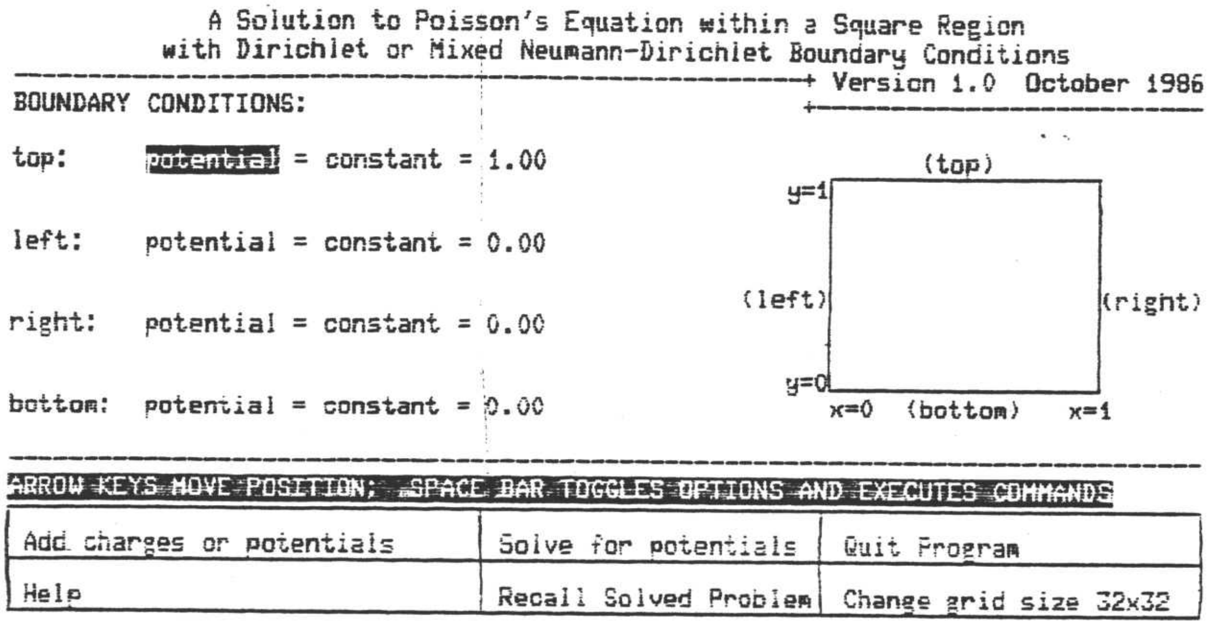
If the user is on the graphics editor screen, there is also a help screen available.



```
This is the graphics editor.  It will allow you to specify the
placement and values of the following (inside the square problem
region)
          1)line charges
          2)rods of constant potential
          3)sheets of constant potential

     QUANTITY         | SYMBOL ON SCREEN |         VALID RANGE OF VALUES
----------------------+------------------+--------------------------------------
                      |                  |
line charge           |   a triangle     | 1.0e-12 <= | Coulomb/z | <= 1.0e-10
                      |                  |
equipotential rod     |    a square      |    -10.0 <= voltage <= 10.0
                      |                  |
equipotential sheet   |     a line       |    -10.0 <= voltage <= 10.0



Hit the SPACE BAR to return to where you were -->
```

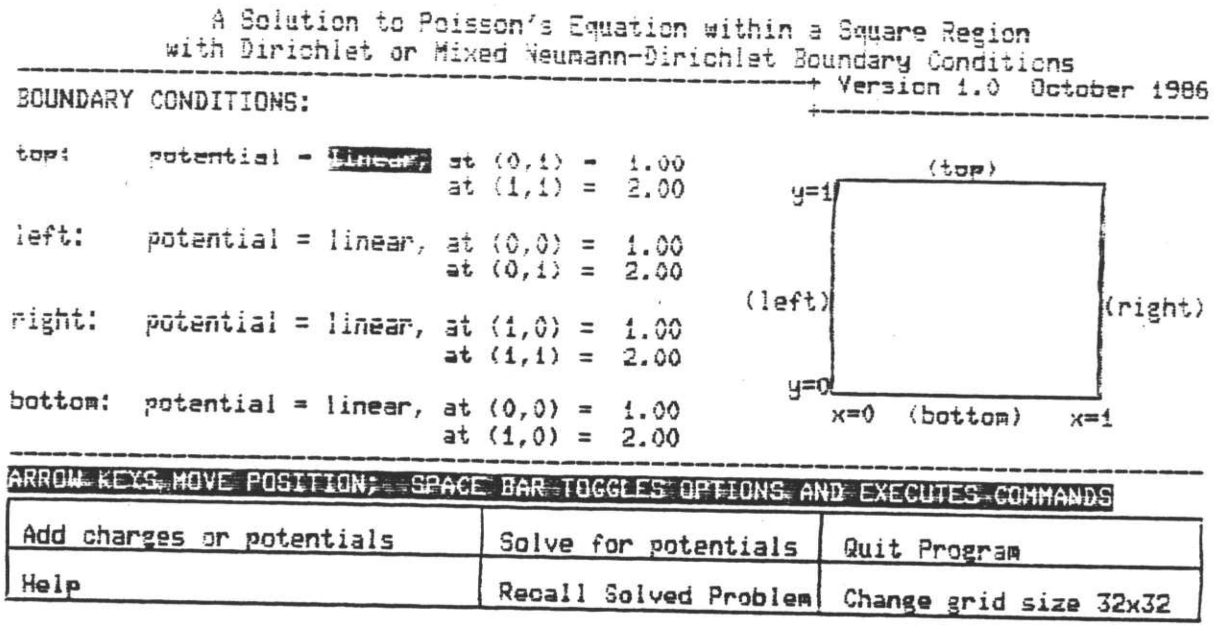
<u>Figure</u> <u>2</u>-<u>3</u>: "Help" message for graphics editor.

The problem region and its boundary conditions are much more self-explanatory, and are less ambigious than before. Contrast Figures (2-4), (2-5), (2-6), (2-7),and (2-8) which their old counterparts [see Appendix B].



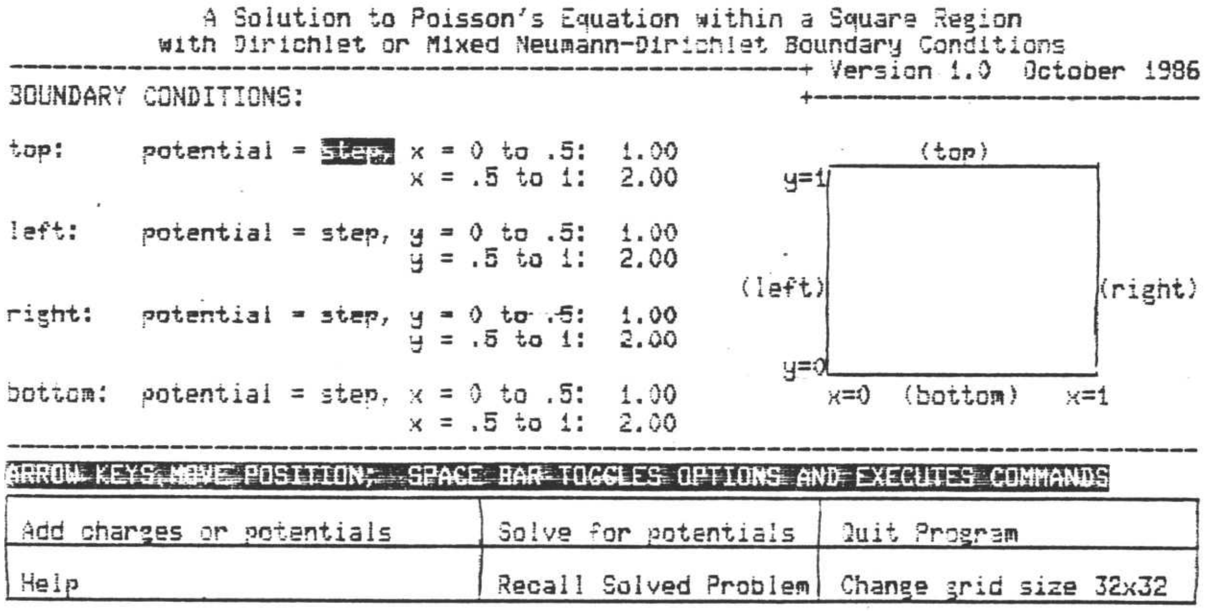
```
        A Solution to Poisson's Equation within a Square Region
      with Dirichlet or Mixed Neumann-Dirichlet Boundary Conditions
-----------------------------------------------------+ Version 1.0  October 1986
BOUNDARY CONDITIONS:                                 +------------------------

top:     potential = constant = 1.00                          (top)
                                                     y=1
left:    potential = constant = 0.00
                                                (left)                (right)
right:   potential = constant = 0.00
                                                     y=0
bottom:  potential = constant = 0.00                 x=0  (bottom)   x=1
--------------------------------------------------------------------------------
ARROW KEYS MOVE POSITION;  SPACE BAR TOGGLES OPTIONS AND EXECUTES COMMANDS
| Add charges or potentials | Solve for potentials  | Quit Program            |
| Help                      | Recall Solved Problem | Change grid size 32x32  |
```

Figure 2-4: The new display: constant boundaries.



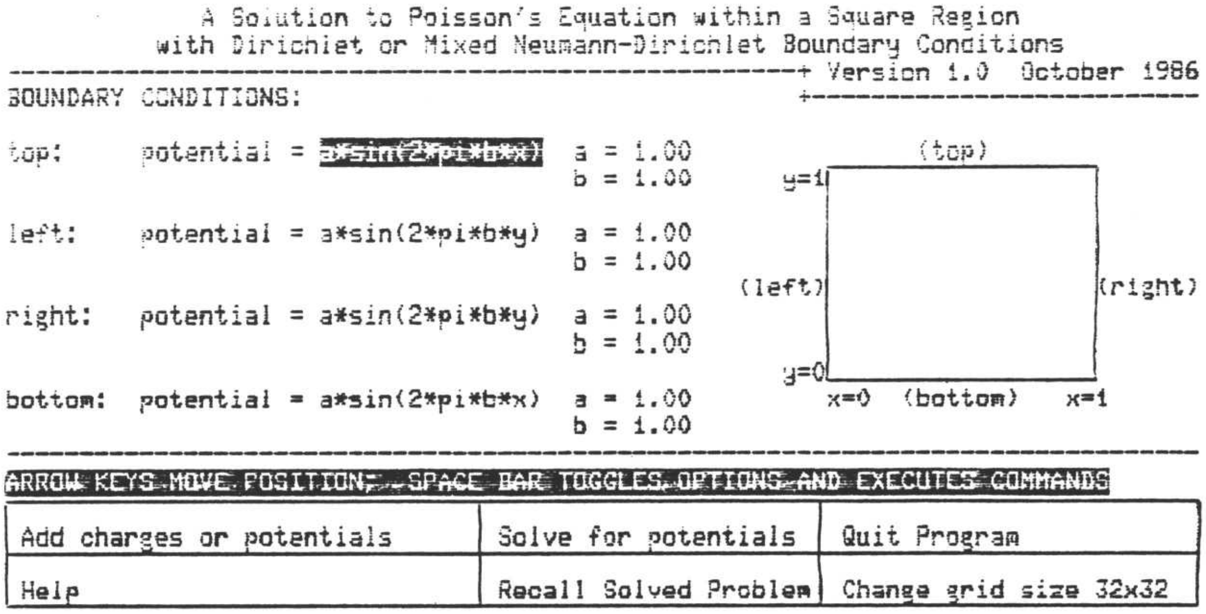
```
        A Solution to Poisson's Equation within a Square Region
      with Dirichlet or Mixed Neumann-Dirichlet Boundary Conditions
-----------------------------------------------------+ Version 1.0  October 1986
BOUNDARY CONDITIONS:                                 +------------------------

top:     potential = linear, at (0,1) =  1.00                 (top)
                             at (1,1) =  2.00        y=1

left:    potential = linear, at (0,0) =  1.00
                             at (0,1) =  2.00
                                                (left)                (right)
right:   potential = linear, at (1,0) =  1.00
                             at (1,1) =  2.00
                                                     y=0
bottom:  potential = linear, at (0,0) =  1.00        x=0  (bottom)   x=1
                             at (1,0) =  2.00
--------------------------------------------------------------------------------
ARROW KEYS MOVE POSITION;  SPACE BAR TOGGLES OPTIONS AND EXECUTES COMMANDS
| Add charges or potentials | Solve for potentials  | Quit Program            |
| Help                      | Recall Solved Problem | Change grid size 32x32  |
```

Figure 2-5: The new display: linear boundaries.



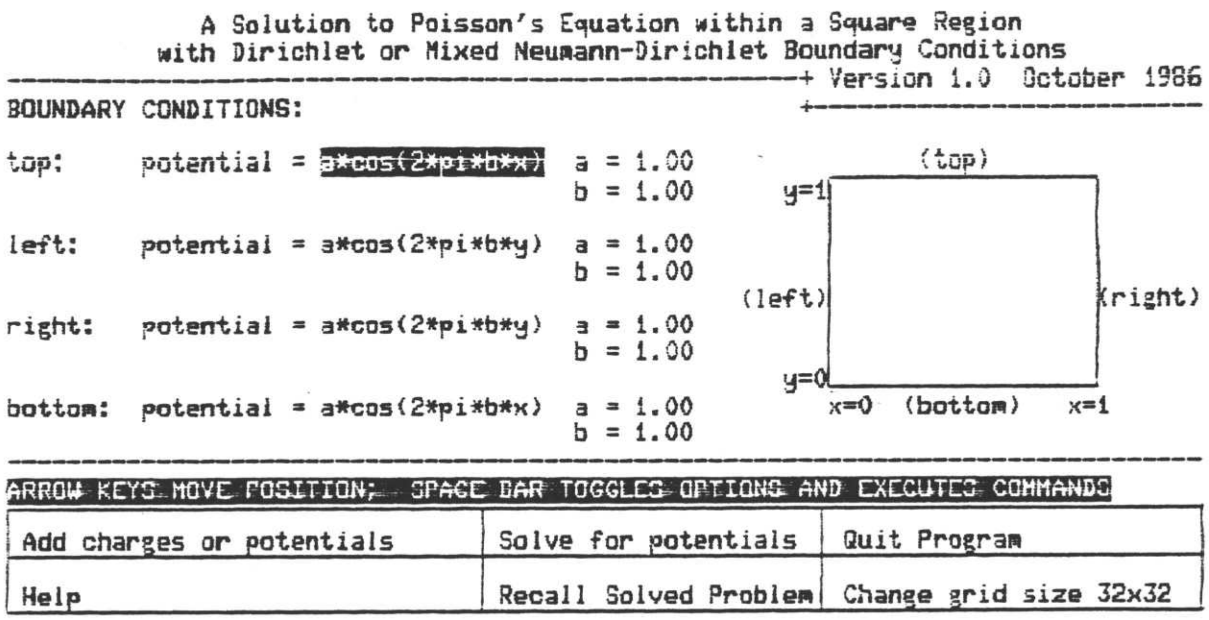
```
          A Solution to Poisson's Equation within a Square Region
        with Dirichlet or Mixed Neumann-Dirichlet Boundary Conditions
---------------------------------------------------+ Version 1.0  October 1986
BOUNDARY CONDITIONS:                               +-----------------------

top:      potential = step, x = 0 to .5:  1.00              (top)
                            x = .5 to 1:  2.00       y=1
left:     potential = step, y = 0 to .5:  1.00
                            y = .5 to 1:  2.00    (left)              (right)
right:    potential = step, y = 0 to .5:  1.00
                            y = .5 to 1:  2.00
                                                     y=0
bottom:   potential = step, x = 0 to .5:  1.00          x=0  (bottom)   x=1
                            x = .5 to 1:  2.00
------------------------------------------------------------------------------
ARROW KEYS MOVE POSITION;  SPACE BAR TOGGLES OPTIONS AND EXECUTES COMMANDS
| Add charges or potentials | Solve for potentials  | Quit Program            |
| Help                      | Recall Solved Problem | Change grid size 32x32  |
```

Figure 2-6: The new display: step boundaries.



```
          A Solution to Poisson's Equation within a Square Region
        with Dirichlet or Mixed Neumann-Dirichlet Boundary Conditions
---------------------------------------------------+ Version 1.0  October 1986
BOUNDARY CONDITIONS:                               +-----------------------

top:      potential = a*sin(2*pi*b*x)  a = 1.00             (top)
                                       b = 1.00      y=1
left:     potential = a*sin(2*pi*b*y)  a = 1.00
                                       b = 1.00   (left)              (right)
right:    potential = a*sin(2*pi*b*y)  a = 1.00
                                       b = 1.00
                                                     y=0
bottom:   potential = a*sin(2*pi*b*x)  a = 1.00         x=0  (bottom)   x=1
                                       b = 1.00
------------------------------------------------------------------------------
ARROW KEYS MOVE POSITION;  SPACE BAR TOGGLES OPTIONS AND EXECUTES COMMANDS
| Add charges or potentials | Solve for potentials  | Quit Program            |
| Help                      | Recall Solved Problem | Change grid size 32x32  |
```

Figure 2-7: The new display: sine boundaries.



```
           A Solution to Poisson's Equation within a Square Region
         with Dirichlet or Mixed Neumann-Dirichlet Boundary Conditions
--------------------------------------------+ Version 1.0  October 1986
BOUNDARY CONDITIONS:                        +-----------------------

top:     potential = a*cos(2*pi*b*x)  a = 1.00          (top)
                                      b = 1.00    y=1

left:    potential = a*cos(2*pi*b*y)  a = 1.00
                                      b = 1.00
                                              (left)           (right)
right:   potential = a*cos(2*pi*b*y)  a = 1.00
                                      b = 1.00
                                                  y=0
bottom:  potential = a*cos(2*pi*b*x)  a = 1.00      x=0  (bottom)   x=1
                                      b = 1.00
-----------------------------------------------------------------------
ARROW KEYS MOVE POSITION;  SPACE BAR TOGGLES OPTIONS AND EXECUTES COMMANDS
| Add charges or potentials | Solve for potentials  | Quit Program          |
| Help                      | Recall Solved Problem | Change grid size 32x32 |
```

Figure 2-8: The new display: cosine boundaries.

It is now much easier to move the high-lighted region. Instead of hunting around the keyboard for the "u", "d", "l", and "r" keys, the user now uses the cluster of 4 arrow keys on the right hand side of the keyboard.

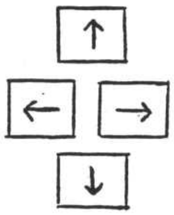
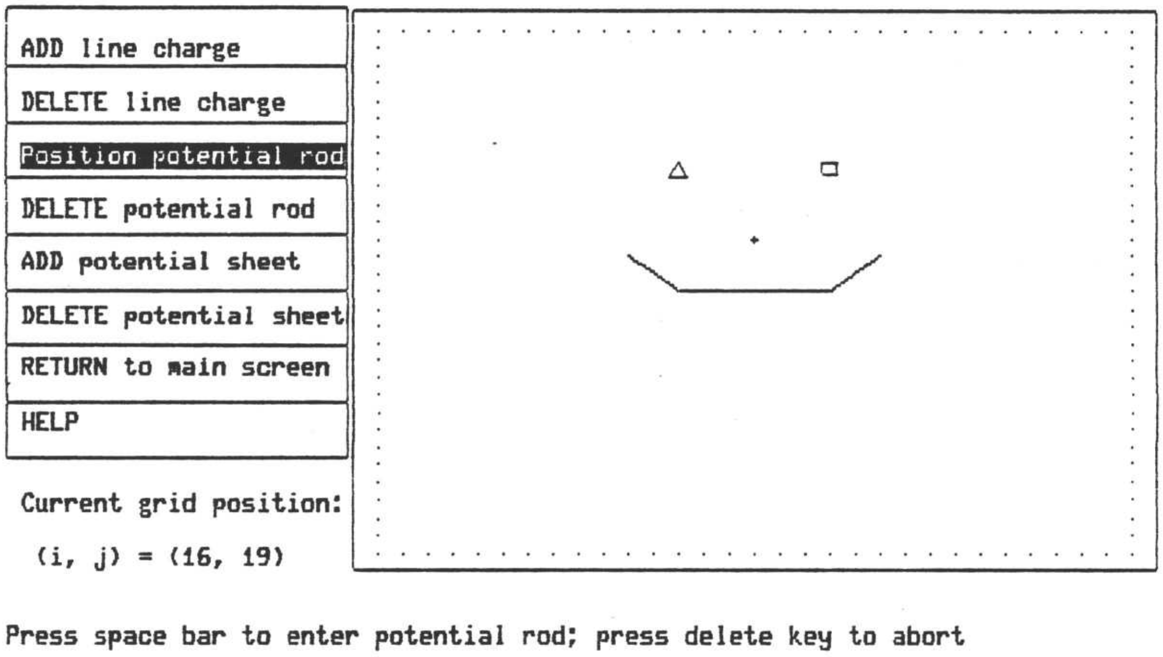


Figure 2-9: Use arrow keys to move high-lighted region.

2.2 The MAIN IMPROVEMENT of the new interface.

The most important improvement that the new interface has is a graphics editor. [Refer to Appendix B for the old method of entering data.]



Current grid position:
(i, j) = (16, 19)

Press space bar to enter potential rod; press delete key to abort

Figure 2-10: The new way to enter data.

In two dimensions, a line charge looks like a point on the terminal screen. A rod of constant potential also looks like a point, and a sheet of constant potential looks like a line. Therefore the user must pick one point (in the interior of the problem region) to specify the position of a line charge or a rod of constant potential. A line is uniquely determined by its two endpoints, so the user must specify two points to position a sheet of constant potential. To specify a point, the user doesn't even need to know its (x,y) coodinates; he merely moves a cross-hair sight about the screen (using the 4 arrow keys), and then hits the space bar when he reaches the desired location. A prompt appears at the bottom of the screen, and asks the user to type in the value of the line charge or the potential.

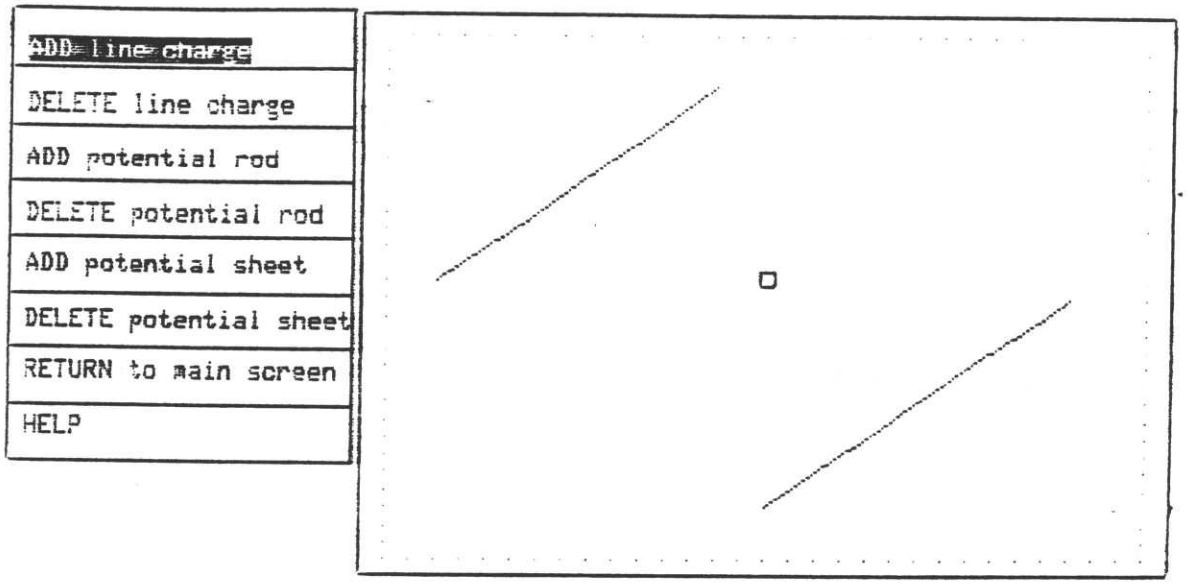
The editor won't let the user get himself into trouble. For example, if the user tries to put a line charge on top of a potential rod, he will be told that there is already a potential rod at that location, and that he should pick another location for the line charge.

The editor also checks to be sure that all sheets of constant potential are either horizontal, vertical, or at a 45-degree angle, as required by "Poisson".

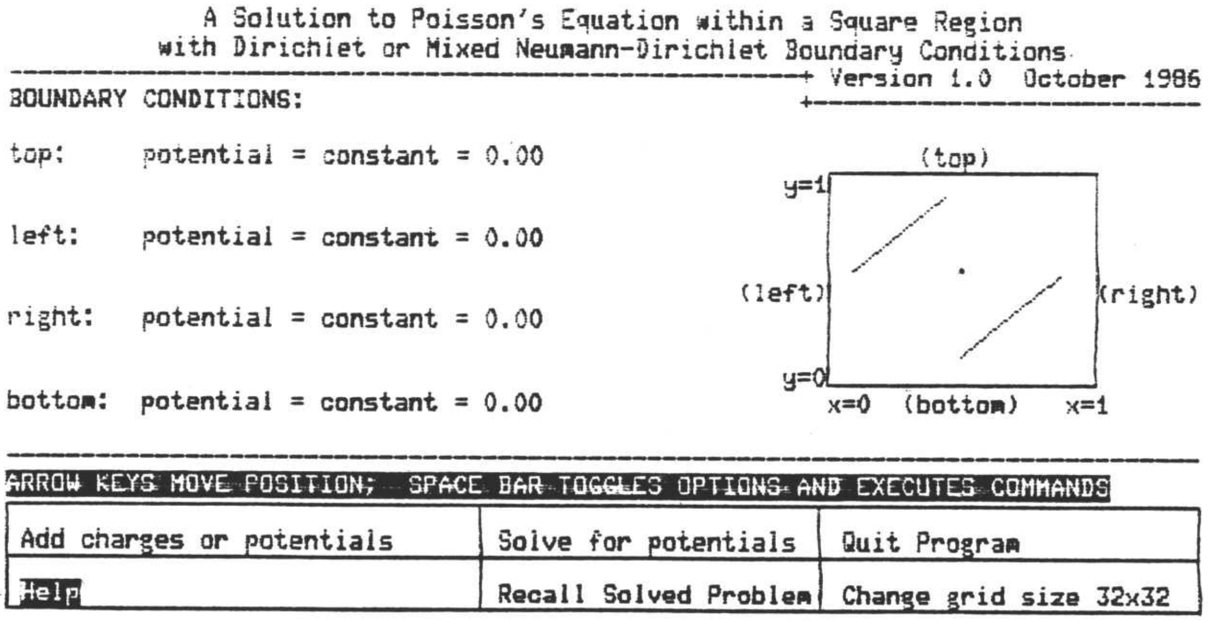
The editor always checks the size of the values that the user enters for the potentials and the charges. If the user attempts to enter a value that is outside the allowable range, then the editor will say so. It will also tell the user what the valid range of values is, and will allow the user to re-enter the value.

Another extremely important feature is the abort key. At any time, the user can abort adding or deleting something from the problem region. All he has to do is hit the delete key.

Another feature of the new interface which makes things a lot more clear is the fact that the problem which is entered in the graphics editor screen is drawn (in miniature) in the small problem box on the main screen [see Figure 2-11].



Problem is entered in the graphics editor.



```
            A Solution to Poisson's Equation within a Square Region
         with Dirichlet or Mixed Neumann-Dirichlet Boundary Conditions
-------------------------------------------------+ Version 1.0  October 1986
BOUNDARY CONDITIONS:                             +-------------------------

top:     potential = constant = 0.00

left:    potential = constant = 0.00

right:   potential = constant = 0.00

bottom:  potential = constant = 0.00
```

Problem then appears in problem box on the main screen.

Figure 2-11: Entering data.

This is a really neat feature. With the old interface, the interior of the problem box on the main screen remained empty after the user entered line charges or potentials inside it. This was quite disconcerting; it made the user wonder if the stuff he entered in the interior of the problem box got accepted by "Poisson" or not.

Another nice thing about the new editor is that the line charges, rods of constant potential, and sheets of constant potential are drawn when the contours are plotted. The old interface just drew the contours.

Note that the symbol for a line charge is a triangle, and the symbol for a rod of constant potential is a square. This scheme was chosen (instead of drawing two different colored squares) so that "Poisson" would not be restricted to color terminals.

Chapter 3

Programming considerations.

3.1 Organization of major functional modules.

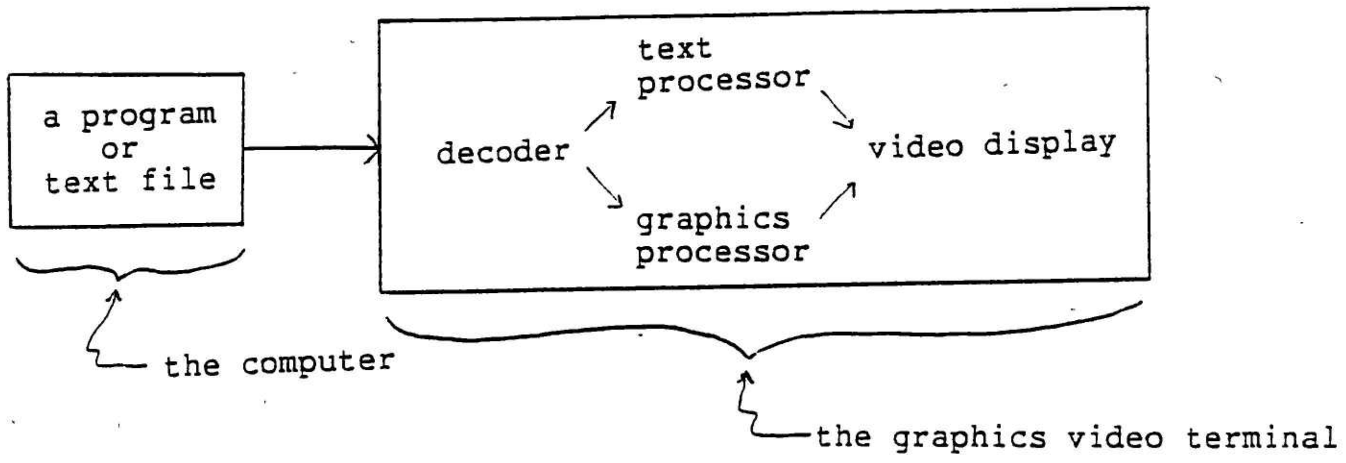
"Poisson" is written in both C and FORTRAN77. Three major modules make up 99% of the source code. These files are compiled separately, and are then linked together with the other compiled files to form the executable code. The three major modules are:

```
Module 1 The user-interface and the graphics editor.
           ---written in C, by Ted C. Johnson.
           ---this is what "talks" to the user, and
              passes the data relevant to the ES
              problem to Module 2.

Module 2 The "engine", i.e., the code which does all
              of the mathematical computations necessary
              to solve Poisson's equation.
           ---written in FORTRAN77, by Denise Barnett.

Module 3 The contour plotter.
           ---this is written in C, by various people.
              Robert Brawer is supporting it.
```

"Poisson" was implemented on a DEC (Digital Equipment Corporation) VAX 11/750, which is one of the minicomputers used by MIT's Project Athena. See Appendix C for the hardware required to run "Poisson".

3.2 Graphics introduction: how most computers deal with graphics.

All video terminals support text display. Some video terminals (graphics terminals) have additional hardware inside of them which enables them to display graphics as well as text. The set up for video terminals which support graphics as well as text is shown below:



<u>Figure 3-1</u>: How a terminal handles graphics and text.

The basic algorithm for displaying graphics and text has 3 steps:

<u>Step 1</u>. Text characters and control characters (both of which are represented via the 8-bit ASCII code, see Appendix G)[2] are sent from the computer to the graphics

---

[2] There are 128 ASCII characters, numbered 0 to 127. Numbers 33 to 126 are text characters; the others are control characters [see Appendix G].

terminal. This is done either by having a program send characters to the terminal, or by the user sending the contents of a text file to the terminal (on a UNIX operating system, this would be done with the "cat" command, e.g., "cat graphics_cmd_file").

Step 2. Inside the terminal, a decoder decides if the character is a text character or a graphics control character. The decoder then sends the character to the proper place to be processed.

Step 3. Text and graphics show up on the user's graphics terminal.

3.3 The graphics systems used in "Poisson".

"Poisson" uses two different graphics systems: ReGIS [1] and Penplot [2]. ReGIS (Remote Graphics Instruction Set) is a set of graphics commands which only work on DEC VT240 terminals. It is extremely flexible and fairly fast, BUT it only works on one particular type of terminal. Penplot is a set of graphics commands which are supposed to be terminal-independent; i.e., in theory, Penplot will work on ANY video terminal which has graphics capabities. It does this by consulting a huge database which has a file of ALMOST every video terminal ever made, and the proper control characters for the graphics commands for that terminal. In reality, Penplot is not terminal-independent; for example, it won't work on the new DEC VAXstation 100 terminals.

3.3.1 The ReGIS graphics system.

ReGIS is a DEC graphics system which only works with certain hardware [see Appendix C]. It allows the programmer to write code to draw points, lines, and circles on the screen, in four different colors (red, green, blue, and black).

3.3.2 The Penplot graphics system.

Penplot is used only for the display of the equipotential contour lines and the electric field lines. ReGIS could have been used to do this, but the author of the contour plotting module built Penplot into the contour plotter. See Appendix E for more information on the contour plotter.

3.4 Major concepts in my code.

The following is an explanation of the rationale behind a few of the major design decisions made while developing and implementing the user-interface for the Poisson-solver "Poisson".

The "Poisson" program uses three screens (not counting the two help screens and the introductory text screen). The first, the main screen, is the screen that the user has the most interaction with. The second screen is the graphics editor, which the user uses to enter line charges, rods of constant potential, and sheets of constant potential into the interior of the problem region. The third screen is the one where the contours are plotted.

The most fundamental decision was the choice of making "Poisson" be menu-driven rather than command-line driven. In other words, all commands are entered by selecting one command from a displayed menu of commands, rather than by the more conventional way of having the user type in commands to a prompt. A command-line driven interface would have been MUCH easier to design and to implement, and it has the additional advantage that it's extremely easy to add more commands to such an interface. A menu-driven interface is much harder to design, takes up more of the terminal's screen space, is harder to implement, and it can be difficult to add more commands to.

Then why pick a menu-driven interface? I picked a menu-driven interface because the main users of the "Poisson" program are going to be students, who have neither the time nor the inclination to spend a lot of time learning how to use a program. Thus, the primary goal for this interface was that FOR THE FIRST-TIME USER IT BE EXTREMELY EASY AND INTUITIVE TO USE. The user interface had to be "intuitively obvious, to the most casual user". A menu-driven interface is MUCH more intuitive, is much easier to use, and is much less prone to user-error than is a command-line driven interface. First of all, there are no commands to memorize (or to forget, or to mistype), because they're all right there on the screen in front of the user. All the user has to do is pick one of them. Second, there is no syntax to worry about, because the user never types in any commands. Third, users who can't type aren't inconvenienced by having to hunt for the right letters to spell out long commands. Fourth, commands which require parameters are also more intuitive, because after the user selects one of these commands from the menu, he is prompted for the necessary parameter(s) at the bottom of the screen (e.g., "Enter the voltage-->").

Chapter 4
Implementation documentation.

4.1 INTRODUCTION.

"Poisson" was implemented using both FORTRAN77 and C programming languages. All of the code associated with the user interface is written in C. All of the code which does the actual number crunching (i.e., the numerical algorithms which find a solution to the electrostatic problem) is written in FORTRAN77. The subroutine which calculates and plots the contours is written in C.

4.2 ORGANIZATION OF FILES.

The user interface is implemented as a subroutine that is called from the FORTRAN77 program [see the file "poisson.f" in Appendix K].

Constants used by the user interface are in the file called "defs" (which stands for "definitions") [see Appendix K].

Structures used by the user interface are defined in the file "structures.c" [see Appendix K].

The ReGIS terminal-dependent graphics subroutines are in the file "regis_subrs.c" [see Appendix K].

All of the structures and arrays used by the user interface are initialized by a huge subroutine called set_default_values(), which is defined in the file "db_init.c" [see Appendix K].

4.3 OVERVIEW.

The general flow of control between the three major modules which make up "Poisson", i.e., (1)the user interface (edit_(...)), (2)the Poisson-solver's number cruncher, and (3)the contour plotter, is shown in the following flow chart, Figure 4-1. This figure shows how these three modules interact with each other.

What follows is an in-depth explanation of the implementation of the user interface. The contour plotter and the Poisson-solver module will be referred to where appropriate. Source code for all three modules is located in Appendices J and K.

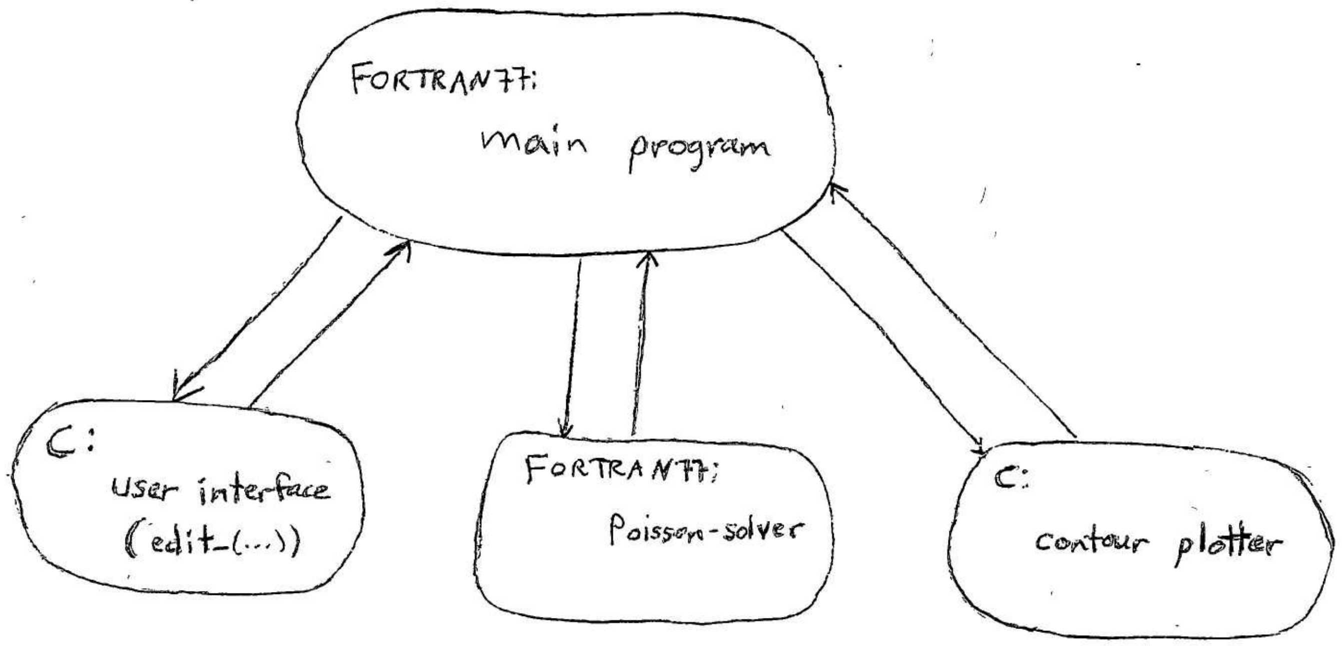
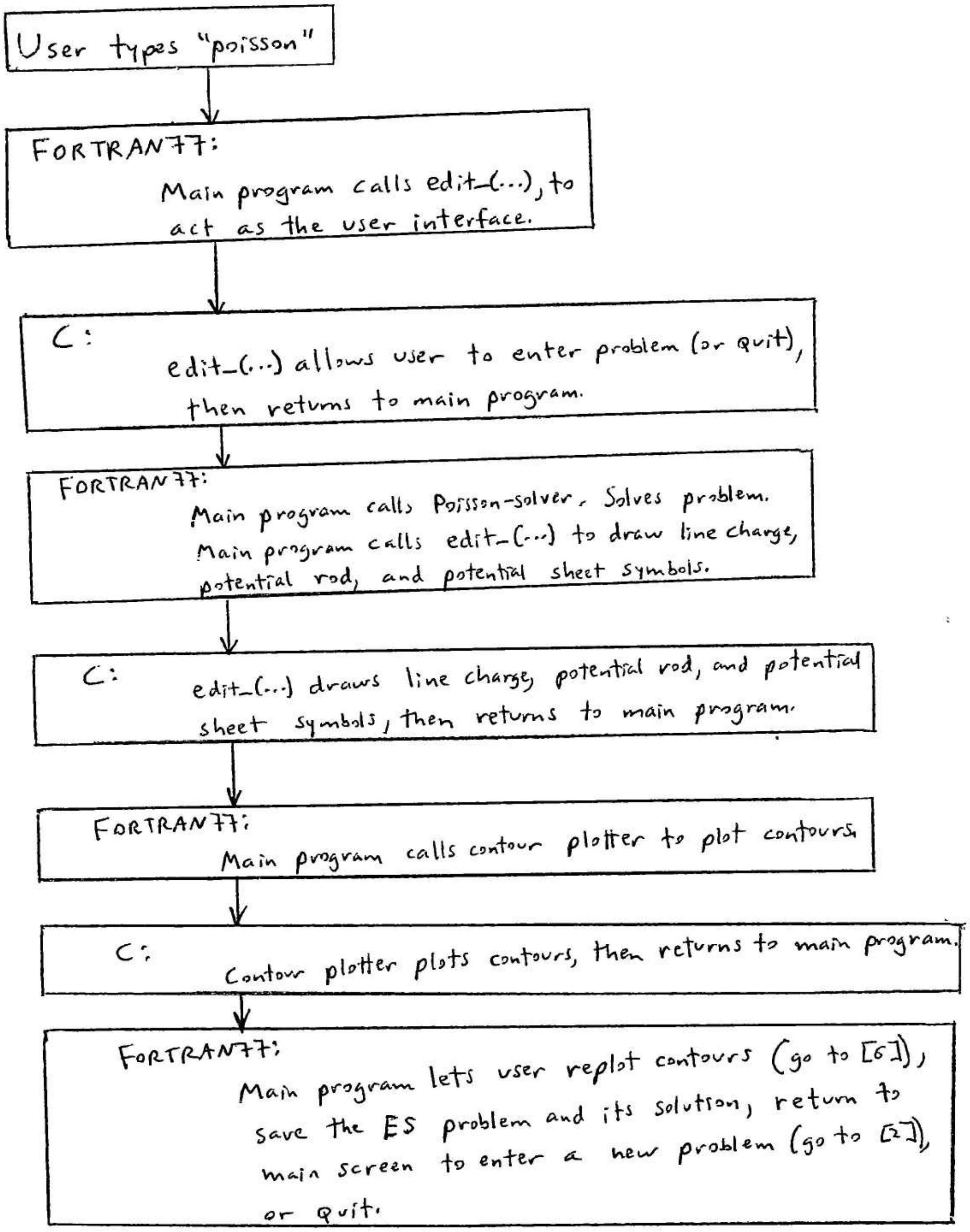


Figure 4-1: Flow of control in "Poisson".



User types "poisson"

FORTRAN77:
Main program calls edit_(...), to act as the user interface.

C:
edit_(...) allows user to enter problem (or quit), then returns to main program.

FORTRAN77:
Main program calls Poisson-solver. Solves problem.
Main program calls edit_(...) to draw line charge, potential rod, and potential sheet symbols.

C:
edit_(...) draws line charge, potential rod, and potential sheet symbols, then returns to main program.

FORTRAN77:
Main program calls contour plotter to plot contours.

C:
Contour plotter plots contours, then returns to main program.

FORTRAN77:
Main program lets user replot contours (go to [6]), save the ES problem and its solution, return to main screen to enter a new problem (go to [2]), or quit.

Figure 4-2: Flow of control in "Poisson".

The implementation of the user interface is explained below. It is divided into two major sections: the main screen and the graphics editor.

Refer to section 4.6 for an explanation of the various data structures, arrays, and arrays of data structures which are used by the user interface.

4.3.1 How the user interface is called by the FORTRAN77 program.

The C subroutine which is called from the FORTRAN77 program is called edit_(...) [see Appendix K, file "fneditor.c", page 2]. It is called for two quite different functions. The first is to act as a user interface, i.e., to let the user specify a problem. The second is to draw the line charge symbols (triangles), the potential rod symbols (squares), and the potential sheet symbols (lines) right before the contour plotter plots the contours. This is all accomplished by the subroutine do teds stuff() [see Appendix K, file "fneditor.c", pages 72-73].

Which of these two functions edit_(...) does is determined by the iflag1 and iflag2 flags passed in to edit_(...). If iflag1 is 1, then edit_(...) is used to plot the line charge, potential rod, and potential sheet symbols. If iflag1 is -1 and iflag2 is 1, then edit_(...) is used as the user interface.

4.4 MAIN SCREEN DOCUMENTATION.

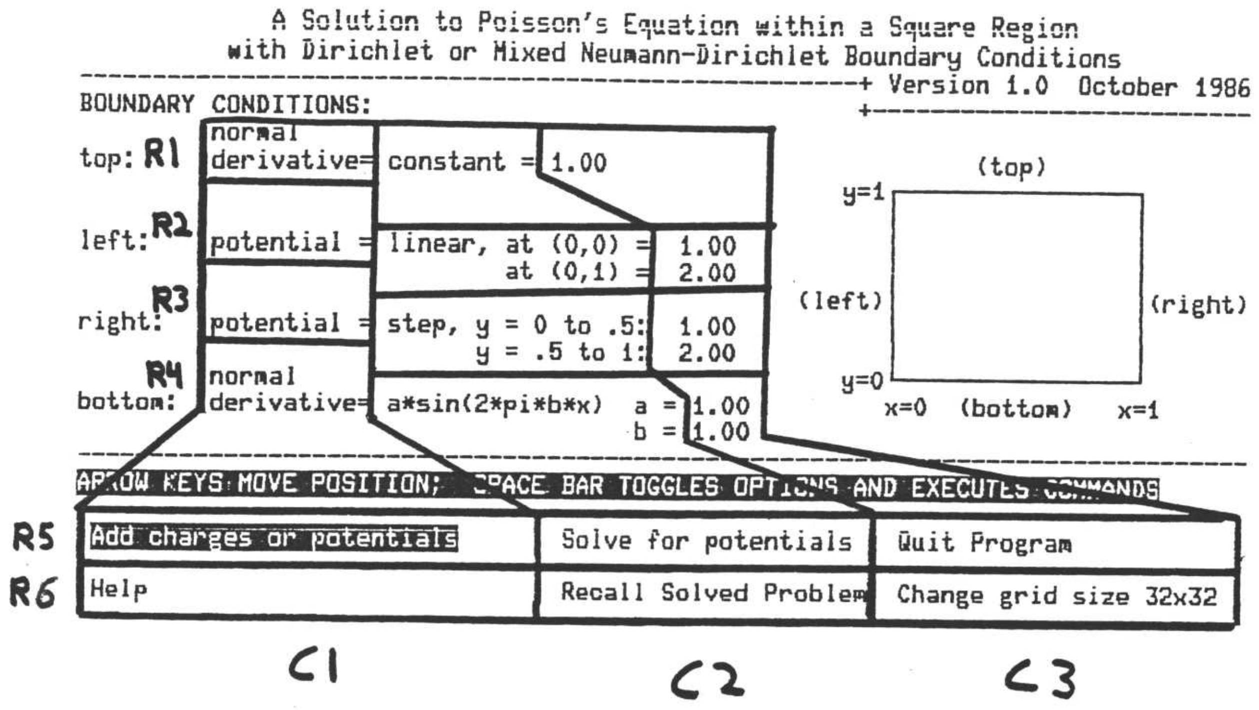
4.4.1 Main screen: display management.

The display of text on the main screen is accomplished by using two subroutines:

```
display(row_num, column_num, want_inverse)
erase(row_num, column_num)
```

There are two coordinate systems that we are concerned about when we are displaying text. The first is the text SCREEN coordinate system. The terminal screen is 24 ASCII characters tall by 80 ASCII characters wide. The text screen coordinate system divides the screen up into an 80 by 24 grid, where the upper left hand corner of the screen is at position (1,1), and the lower right hand corner of the screen is at position (24, 80).

The second coordinate system is one that is artifically imposed on the main screen's display. It is called the text GRID coordinate system. It is used to keep track of the portions of the main screen where the text is liable to change (e.g., the different types of functions available for the boundaries of the problem region). It divides the active portions of the screen into a grid that is six rows (labelled R1 to R6) tall by three columns (labelled C1 to C3) wide. See Figure 4-3.



```
           A Solution to Poisson's Equation within a Square Region
        with Dirichlet or Mixed Neumann-Dirichlet Boundary Conditions
----------------------------------------------------+ Version 1.0  October 1986
BOUNDARY CONDITIONS:                                +-------------------------
        normal
top:    derivative= constant = 1.00                          (top)
                                                     y=1
left:   potential = linear, at (0,0) =  1.00
                           at (0,1) =  2.00
                                                  (left)               (right)
right:  potential = step, y = 0 to .5:  1.00
                          y = .5 to 1:  2.00
        normal                                       y=0
bottom: derivative= a*sin(2*pi*b*x)  a = 1.00        x=0   (bottom)    x=1
                                     b = 1.00
-------------------------------------------------------------------------------
ARROW KEYS MOVE POSITION;  SPACE BAR TOGGLES OPTIONS AND EXECUTES COMMANDS
Add charges or potentials      Solve for potentials    Quit Program
Help                           Recall Solved Problem   Change grid size 32x32
```

Figure 4-3: Text grid coordinate system.

4.4.1.1 Explanation of display(...).

The display(...) subroutine [see Appendix K, file "fneditor.c", pages 30-33] is actually a database. It contains ALL of the text that is displayed on the main screen, with the exception of the title that appears at the top of the screen. The display(...) subroutine contains the text for all the different options (e.g., it contains the text strings: "potential", "normal derivative", "linear", "step", "constant", etc.). It also contains the text for all of the menu items (e.g., it contains the text strings: "Add charges or potentials", "Solve for potentials", "Help", "Quit Program", etc.).

display(...) is called with the text grid coordinates of the location where text is to be displayed; it then looks in the options[][] database to see which of several text strings are meant to be displayed at this text grid position. For instance, in column two (text grid positions (R1,C2), (R2,C2), (R3,C2), and (R4,C2)), the text can be one of two things, either "potential =" or "normal derivative =". The way display(...) knows which of these to use is by consulting the options[][] array, and seeing what option was selected for that text grid location.

The next step is to see what text screen coordinates correspond to these text grid coordinates. This is done by consulting the db[] database, which contains the text screen coordinates for all of the text that is displayed on the main screen.

There are two modes in which the text can be displayed: normal video, and inverse video. In normal video mode, the text appears as white characters on a black background. In inverse video mode, the text is black and the background is white. Inverse video is used to draw the user's attention to a certain area of the screen. "Poisson" uses this method, instead of a cursor, to indicate to the user where the next action will take place. If the want_inverse argument to display(...) is YES, then the text will be typed in inverse video mode. Otherwise the flag is NO, and the text is typed in normal video mode.[3]

[3] YES and NO are constants which are defined in the "defs" file [see Appendix K]. The value of YES is 1; the value of NO is 0.

4.4.1.2 Explanation of erase(...).

The counterpart to the display(...) subroutine is called erase(...) [see Appendix K, file "fneditor.c", pages 28-29]. It is also a database: it knows how many blank spaces to type out to erase any of the text grid locations, no matter what option is being displayed.

Why is the erase(...) subroutine needed? Let's look at a typical scenario. The user has just toggled the "linear" option to the "step" option, for the top boundary of the problem box (i.e., text grid location (R1,C3)). If we just use display(...) to type "step" where "linear" used to be shown, then what will actually appear on the screen is "stepar", i.e., only the first four characters of "linear" will be overwritten by the text string "step". The way to fix this problem is to erase the text string "linear" first. Erasing text is done by writing over it with blank spaces. Different numbers of blank spaces are needed, depending on how much text is to be erased.

The way the erase(...) subroutine works is that you pass it the text grid coordinates, i.e., the (row_num, column_num), of the text grid location that you want erased. erase(...) then consults the options[][] database, to see what option is being displayed at this text grid location. Then the db[] database is consulted, to see what text screen coordinates correspond to the text grid location (row_num, column_num). Finally, <u>erase(...)</u> has all the information it needs, so it types out the correct number of blank spaces to erase the text that is currently being displayed at text grid location (row_num, column_num).

4.4.1.3 Usage of <u>display(...)</u> and <u>erase(...)</u>.

<u>display(...)</u> and <u>erase(...)</u> are used in two separate situations. The first is when the user toggles from one option to another. The program then uses <u>erase(...)</u> to erase the current option, and uses <u>display(...)</u> to type out the text for next option.

The second time these subroutines are used is when the user moves the high-lighted region from one place to another. The program uses <u>erase(...)</u> to erase the current high-lighted text grid location, and uses <u>display(...)</u> to re-type it, this time in normal video mode. The program then uses <u>erase(...)</u> to erase the text of the text grid location that the user wants to move to, and re-types that text (in inverse video mode) using <u>display(...)</u>.

4.4.2 Main screen: handling user interaction.

"Poisson" lets the user move the high-lighted region to any of the 18 text grid coordinate locations (6 columns x 3 rows = 18 locations) [see Figure 4-3], via the four arrow keys on the right hand side of the keyboard. This is done via the monitor_keyboard(...) subroutine [see Appendix K, file "fneditor.c", page 12].

Wrap-around is permitted; i.e., if the high-lighted region is at the leftmost side of the screen and the user hits the left arrow key, then the high-lighted region will appear at the right side of the screen. Wrap-around to the left, to the right, over the top, and under the bottom of the screen are all permitted.

The program ignores all keys except for the four arrow keys and the space bar. No action is taken until the user hits the space bar.

4.4.2.1 Menu item selection.

When the user hits the space bar, the program checks the current text grid row number (which is kept track of by the variable cur_row) to see if the high-lighted region is in the menu. It is in the menu if the current row number is R5 or R6 [see Figure 4-3]. If it is, then the menu_handler(...) subroutine [see Appendix K, file "fneditor.c", page 14] is called.

The menu_handler(...) determines (via the variables cur_col and cur_row, which hold the current text grid row and column numbers) which menu item the user selected.

Menu item: "Add charges or potentials"

If the user selected the "Add charges or potentials" menu item, (row,col) = (R5,C1), then the menu_handler(...) calls the invoke_screen2() subroutine [see Appendix K, file "fneditor.c", page 17]. This subroutine erases the screen, and conjures up the graphics editor screen. See section 4.5 for information on the implementation of the graphics editor.

Menu item: "Solve for potentials"

If the user selected the "Solve for potentials" menu item, (row,col) = (R5,C2), then the menu_handler(...) calls the crunch() subroutine [see Appendix K, file "fneditor.c", page 18]. This subroutine makes the global variable quit equal YES, and the main loop [see Appendix K, file "fneditor.c", page 4]

```
while (quit == NO) {
        monitor_keyboard();
}
```

is exited. Then the arrays passed in to edit_(...) [see

Appendix K, file "fneditor.c", page 2] from the FORTRAN77 main program [see Appendix J, file "poisson.f"] are filled with the voltages and the line charges which describe the problem that the user entered. The edit_(...) subroutine is then exited, and control is returned to the FORTRAN77 program which called it [see Appendix J, file "poisson.f"].

Next, the FORTRAN77 modules solve Poisson's equation, with the sources and boundary conditions specified by the user. The FORTRAN77 main program then calls edit_(...), with the variable iflag1 equal to 1. This tells edit_(...) to plot the symbols for the line charges, the potential rods, and the potential sheets. The edit_(...) subroutine is then exited, and control returns to the FORTRAN77 main program. The main program then calls the contour plotter [see Appendix J, file "poisson.f"], and passes it the solution to the ES problem. The contour plotter [see Appendix E] plots the contours, and returns control to the FORTRAN77 main program. If the user wants to return to the main screen (to do another problem), then edit (...) is called again, this time with iflag1 equal to -1 and iflag2 equal to 1 (which tells edit_(...) to act as a user interface).

Menu item: "Quit Program"

If the user selected the "Quit Program" menu item, (row,col) = (R5,C3), then the menu_handler(...) calls the break_out() subroutine [see Appendix K, file "fneditor.c", pages 18-19]. This subroutine asks the user, "Do you really want to quit (y/n)?" If the user types "n" or "N", then break_out() clears the screen, and terminates the "Poisson" program. Since we want to terminate the *entire* program from inside a subroutine, and not from inside the main program (which is the usual way of doing it), we have to terminate the program in an inelegant fashion; this is done by using the C exit() function [see Appendix K, file "fneditor.c", page 19].

Menu item: "Help"

If the user selected the "Help" menu item, (row,col) = (R6,C1), then the menu_handler(...) calls the help() subroutine [see Appendix K, file "fneditor.c", pages 17-18]. This subroutine clears the entire screen, and then types a few paragraphs of helpful text. help() tells the user to indicate when he is done reading the help screen and is ready to return to the main screen by hitting the space bar. When the user hits the space bar, the screen is erased and the main screen is redrawn.

Menu item: "Recall Solved Problem"

If the user selected the "Recall Solved Problem" menu item, (row,col) = (R6,C2), then the menu_handler(...) calls the get_problem(...) subroutine [see Appendix K, file "fneditor.c", pages 14-15]. This subroutine prompts the user for a file name, and then reads all the data contained in that file into the following arrays: pr_grid[][], lc_grid[][], ps_grid[][], ps_db[][], pot[][], options[][], and db[].

It may seem odd that saving a problem is NOT one of the options on the menu, yet recalling a problem is. How can you recall a problem if you can't save it in the first place? The answer to this is that "Poisson" doesn't ask the user if he wants to save a problem until AFTER the problem has been solved, and the contours have been plotted. The FORTRAN77 main program [see Appendix J, file "poisson.f"] takes care of asking the user if he wants to save this problem, and if so, what file should the problem be saved in. The saving of a problem is extremely straightforward. The arrays pr_grid[][], lc_grid[][], ps_grid[][], ps_db[][], pot[][], options[][], and db[] are simply written into the file that the user specifies.

Menu item: "Change grid size"

If the user selected the "Change grid size" menu item, (row,col) = R6,C3), then the menu_handler(...) calls the grid_size() subroutine [see Appendix K, file "fneditor.c", page 16]. There are two grid sizes available, 64x64 and 32x32. The larger grid size causes the FORTRAN77 program to compute the solution to Poisson's equation to a higher degree of accuracy than with a 32x32 grid size. This results in smoother contours when the contours are plotted. The trade-off is that doubling the grid size causes the Poisson-solver to take four times as long to solve the problem. Aside from the smoother contours and the longer wait, the larger grid size looks exactly the same to the user. It does NOT enable the user to fit more line charges in the inside of the problem region, or of that nature.

The grid_size() subroutine is very simple. It determines what the current grid size is (by checking the size global variable), and toggles it to the other size. It toggles size between 32 and 64. The subroutine then updates the main screen to indicate the current grid size.

4.4.2.2 Toggling the boundary condition options, and changing the numerical parameters of the boundary condition options.

If the high-lighted region ISN'T in the menu area (which consists of rows R5 and R6), then the user wants to either: (1)toggle between specifying the "normal derivative" of the potential and the "potential", or

(2)toggle between having the normal derivative/potential be constant, a step function, linear, a sine, or a cosine, or (3)change the numerical parameters of one of the five normal derivative/potential functions.

Whenever the space bar is hit and the high-lighted region isn't in the menu, monitor_keyboard(...) calls the subroutine space_bar() [see Appendix K, file "fneditor.c", page 23].

The space_bar() subroutine checks the cur_cul variable to see which of the three text grid columns [see figure 4-3] the high-lighted region is in. What next occurs depends on which column the high-lighted region is in.

If the high-lighted region is in column one:

If space_bar() finds that the high-lighted region is in column one, then it uses the cur_col, cur_row and erase(...) to erase the text for the current high-lighted option (which is either "potential" or "normal derivative"). It then displays the text for the other option (which is either "normal derivative" or "potential") via the display(...) subroutine.

If the high-lighted region is in column two:

If space_bar() finds that the high-lighted region is in column two, then it erases the current option, and draws the next option. The order in which the options are toggled through is: "constant" --> "linear" --> "step" --> "sine" --> "cosine". When it gets to "cosine", it starts over again at "constant".

<u>If the high-lighted region is in column three:</u>

If <u>space_bar()</u> finds that the high-lighted region is in column three, then that means that the user wants to change the numerical parameter(s) associated with that boundary's potential/normal derivative function. This is a bit tricky.

In order to prompt the user for a new value, we must know what option is being used on this boundary. If all the boundaries are linear [see Figure 2-5], then we want to ask the user for the values at the coordinates (0,0), (0,1), (1,0), and (1,1). We need to know what text to put in our prompt, e.g, "Enter new value at (0,1) -->". This is done with the <u>better_get_num()</u> subroutine [see Appendix K, file "fneditor.c", pages 23-28].

<u>Explanation of better_get_num()</u>

<u>better_get_num()</u> checks <u>cur_col</u> and <u>cur_row</u> to see what the current text grid location is. It then checks the <u>options[][]</u> database to see what option is being used at

(cur_row,cur_col). The subroutine then prompts the user for the appropriate values. The arrays lnr_str[] ("LiNeaR STRing") and stp_str[] ("STeP STRing") [see Appendix K, file "init.c"] are used to help better_get_num() construct the proper text strings to prompt the user for the parameter values.

better_get_num() checks to make sure that the data the user types in is numeric. If is isn't, it tells the user "Non-numeric data. Rejected." and does not prompt the user to try again. The user must hit the space bar again if he wants to try again to change this numeric parameter.

After the new numeric value(s) have been received from the user, better_get_num() writes them into the database db[] [see section 4.6], tells the user that the new value(s) have been accepted, and then updates the parameter(s) on the main screen, to reflect the new value(s).

4.5 THE GRAPHICS EDITOR: DISPLAY MANAGEMENT AND USER INTERACTION.

4.5.1 Introduction.

The graphics editor is used to enter line charges, rods of constant potential, and sheets of constant potential into the interior of the problem region. It is invoked from the main screen, via the "Add charges of potentials" menu item. As mentioned earlier, menu_handler(...) then calls the subroutine invoke_screen2() [see Appendix K, file "fneditoir.c", page 17], to set up and run the graphics editor.

The first thing that invoke_screen2() does is set the variable want_secondary_screen to YES. It then erases the screen, and draws the graphics editor screen (via the subroutine draw_secondary_screen() [see Appendix K, file "fneditor.c", page 35]. This subroutine draws the editor menu and the graphics editor grid, as well as any line charges, potential rods, or potential sheets that were previously entered. The following loop is then entered:

```
while (want_secondary_screen == YES)  {
        smonitor_keyboard();
}
```

As soon as this loop is exited, the graphics editor screen is erased and the main screen is redrawn. Any line charges, potential rods, or potential sheets that were entered are mapped into the problem box on the main screen [see Figure 2-11].

Note that all of the subroutines which are used exclusively by the graphics editor are prefixed with the letter "s". This is because the graphics editor was originally called the "Secondary screen".

4.5.2 The graphics editor: user interaction.

The user is allowed to move the high-lighted region up and down the eight-item menu, using the and arrow keys, which are located at the lower right hand side of the VT240 keyboard. Wrap-around over the top and under the bottom of the menu are permitted. The and arrow keys, and all other keys (with the exception of the space bar and the delete key) are ignored.

As on the main screen, the space bar is used to select a menu item. The delete key is used if the user wants to change his mind. For example, the user may select the "ADD line charge" menu item, and then decide that he doesn't really want to add a line charge. Instead of forcing him to add a line charge and then delete it (by selecting "DELETE line charge"), the <u>smonitor_keyboard()</u> subroutine [see Appendix K, file "fneditor.c", pages 36-37] allows him to abort the "ADD line charge" command by hitting the delete key. All of the commands can be aborted, with the exception of the "RETURN to main screen" and "HELP" commands. Aborting a command has no ill effects whatsoever.

Adding a line charge.

Adding a line charge is done via the sadd_lcharge() subroutine [see Appendix K, file "fneditor.c", pages 38-40]. The algorithm for this subroutine is given below [see Algorithm 4-1]. All the user has to do is specify the position of the line charge, and its value (lambda).

This process is as user-proof as possible. sadd_lcharge() checks to make sure that the user doesn't try to put a line charge outside of the problem region, or on a boundary. It doesn't let him put a line charge on top of a potential rod, on top of a potential sheet, or on top of another line charge. When the user is prompted for the line charge's lambda, he is told to re-enter the data if he types in something non-numeric. If the user tries to specify a lambda that is outside the allowable range, he is told what the allowable range is, and is told to re-enter the value of lambda.

ALGORITHM 4-1: ADDING A LINE CHARGE.

```
[1] User selects the "ADD line charge" menu item.
         -Draw cross-hair in center of problem region.
          Tell user "Position line charge."
          Tell user "Press space bar to enter line charges; press delete
                     key to abort."

[2] Did user hit the delete key?
         -if YES, Erase cross-hair.
                  Tell user "Aborted -- add line charge."
                  Return to menu.

         -if NO,  Continue.

[3] Did user hit the ↑, ↓, →, or ← key?
         -if YES, Check to see if this would move the cursor out of the
                  problem region.

                  -if YES, Do nothing.

                  -if NO,  Move the cross-hair one grid unit in the
                           indicated direction.

         -if NO,  Continue.

[4] Did user hit the space bar?
         -if NO,  Go to [2].

         -if YES, [4a] Check the potential rod array:  does a potential
                       rod exist at this (x,y) grid point?

                       -if YES, Tell user "Rod of constant potential is
                                           here.  Try again."
                                Go to [2].

                       -if NO,  Continue.

                  [4b] Check the potential sheet array:  does a potential
                       sheet pass through this (x,y) grid point?

                       -if YES, Tell user "Sheet of constant potential is
                                           here.  Try again."
                                Go to [2].

                       -if NO,  Continue.

                  [4c] Check the line charge array:  does a line charge
                       already exist at this (x,y) grid point?

                       -if YES, Tell user "Line charge is already here.
                                           Try again."
                                Go to [2].

                       -if NO,  Continue.
```

```
[4d] Tell user "Enter lamda for line charge, in
                Coulombs/unit length-->"
     Read in user's input.
     Check:  did user enter non-numeric data?

     -if YES, Tell user "Non-numeric data.  Try again."
              Go to [4d].

     -if NO,  Continue.

[4f] Check:  is the number that the user entered in the
     range of 1.0e-12 <= abs_value(lambda) <= 1.0e-10?

     -if YES, Tell user "Line charge has lambda = X",
              where X is the value the user entered.
              Make line charge array "chosen" flag for
              this grid point equal YES.
              Make line charge array "value" parameter
              for this grid point equal X.
              Return to menu.

     -if NO,  Tell user "Valid range:  1.0e-12 <=
              abs_value(lambda) <= 1.0e-10.  Try again."
              Go to [4d].
```

Deleting a line charge.

Deleting a line charge is done via the sdelete_lcharge() subroutine [see Appendix K, file "fneditor.c", pages 40-41]. The algorithm for this subroutine is given below [see Algorithm 4-2]. All the user has to do is specify the position of the line charge.

This process is also as user-proof as possible. The subroutine first checks to see if any line charges exist. If not, then it tells the user that there aren't any line charges, and then it returns to the menu. The subroutine makes sure that what the user deletes is indeed a line charge, and not a potential rod or a potential sheet.

ALGORITHM 4-2: DELETING A LINE CHARGE.

```
[1] User selects the "DELETE line charge" menu item.

[2] Check the line charge array:  do any line charges exist?
        -if YES, Draw cross-hair in center of problem region.
                 Tell user "Move to line charge."
                 Tell user "Press space bar to delete line charge;
                            delete key to abort."

        -if NO,  Tell user "No line charges exist."
                 Return to menu.

[3] Did the user hit the delete key?
        -if YES, Erase cross-hair.
                 Tell user "Aborted -- delete line charge."
                 Return to menu.

        -if NO,  Continue.

[4] Did user hit the ↑, ↓ , ←, or → key?
        -if YES, Check to see if this would move the cursor out of the
                 problem region.

                 -if YES, Do nothing.

                 -if NO,  Move the cross-hair one grid unit in the
                          indicated region.

        -if NO,  Continue.

[5] Did user hit the space bar?
        -if YES, Check the line charge array:  does a line charge exist at
                 this (x,y) grid point?

                 -if YES, Make line charge array "chosen" flag for this
                          grid point equal NO.
                          Erase line charge from the screen.
                          Tell user "Line charge has been deleted."
                          Return to menu.

                 -if NO,  Tell user "No line charge exists here.
                                     Try again."
                          Go to [3].

        -if NO,  Go to [3].
```

Adding a potential rod.

Adding a potential rod is done via the sadd_prod() subroutine [see Appendix K, file "fneditor.c", pages 42-44]. The algorithm for this subroutine is given below [see Algorithm 4-3]. All the user has to do is specify the position of the potential rod, and its voltage.

This process is identical to that of adding a line charge, so I won't explain the algorithm. See the above explanation for adding a line charge for an explanation.

ALGORITHM 4-3:   ADDING A POTENTIAL ROD.

```
[1] User selects the "ADD potential rod" menu item.
        -Draw cross-hair in center of problem region.
         Tell user "Position potential rod."
         Tell user "Press space bar to enter line charges; press delete
                    key to abort."

[2] Did user hit the delete key?
        -if YES, Erase cross-hair.
                 Tell user "Aborted -- add potential rod."
                 Return to menu.

        -if NO,  Continue.

[3] Did user hit the ↑, ↓ , ← , or → key?
        -if YES, Check to see if this would move the cursor out of the
                 problem region.

                 -if YES, Do nothing.

                 -if NO,  Move the cursor-hair one grid unit in the
                          indicated direction.

        -if NO,  Continue.

[4] Did user hit the space bar?
        -if NO,  Go to [2].

        -if YES, [4a] Check the potential rod array:  does a potential
                      rod exist at this (x,y) grid point?

                      -if YES, Tell user "Rod of constant potential is
                                         already here.  Try again."
                               Go to [2].

                      -if NO,  Continue.

                 [4b] Check the potential sheet array:  does a potential
                      sheet pass through this (x,y) grid point?

                      -if YES, Tell user "Sheet of constant potential is
                                         here.  Try again."
                               Go to [2].

                      -if NO,  Continue.

                 [4c] Check the line charge array:  does a line charge
                      already exist at this (x,y) grid point?

                      -if YES, Tell user "Line charge is here.  Try
                                         again."
                               Go to [2].

                      -if NO,  Continue.
```

```
[4d] Tell user "Enter voltage for potential rod-->"
     Read in user's input.
     Check:  did user enter non-numeric data?

     -if YES, Tell user "Non-numeric data.  Try again."
              Go to [4d].

     -if NO,  Continue.

[4f] Check:  is the number that the user entered in the
     range of -10.0 <= voltage <= 10.0?

     -if YES, Tell user "Potential rod has voltage = X",
              where X is the value the user entered.
              Make potential rod array "chosen" flag for
              this grid point equal YES.
              Make potential rod array "value" parameter
              for this grid point equal X.
              Return to menu.

     -if NO,  Tell user "Valid range:  -10.0 <= voltage
              <= 10.0  Try again."
              Go to [4d].
```

Deleting a potential rod.

Deleting a potential rod is done via the sdelete_prod() subroutine [see Appendix K, file "fneditor.c", pages 44-45]. The algorithm for this subroutine is given below [see Algorithm 4-4]. All the user has to do is specify the position of the potential rod.

This process is identical to that of deleting a line charge, so I won't explain the algorithm. See the above explanation for deleting a line charge for an explanation.

ALGORITHM 4-4: DELETING A ROD OF CONSTANT POTENTIAL.

```
[1] User selects the "DELETE potential rod" menu item.

[2] Check the potential rod array:  do any potential rods exist?
        -if YES, Draw cross-hair in center of problem region.
                 Tell user "Move to potential rod."
                 Tell user "Press space bar to delete potential rod;
                            delete key to abort."

        -if NO,  Tell user "No potential rods exist."
                 Return to menu.

[3] Did the user hit the delete key?
        -if YES, Erase cross-hair.
                 Tell user "Aborted -- delete potential rod."
                 Return to menu.

        -if NO,  Continue.

[4] Did user hit the ↑, ↓ , ←, or → key?
        -if YES, Check to see if this would move the cursor out of the
                 problem region.

                 -if YES, Do nothing.

                 -if NO,  Move the cross-hair one grid unit in the
                          indicated region.

        -if NO,  Continue.

[5] Did user hit the space bar?
        -if YES, Check the potential rod array: does a potential rod exist
                 at this (x,y) grid point?

                 -if YES, Make potential rod array "chosen" flag for this
                          grid point equal NO.
                          Erase potential rod from the screen.
                          Tell user "Potential rod has been deleted."
                          Return to menu.

                 -if NO,  Tell user "No potential rod exists here.
                                     Try again."
                          Go to [3].

        -if NO,  Go to [3].
```

Adding a potential sheet.

Adding a potential sheet is done via the sadd_psheet() subroutine [see Appendix K, file "fneditor.c", pages 45-47] and the spart2_add_psheet(...) subroutine [see Appendix K, file "fneditor.c", pages 47-51]. The algorithms for both of these subroutines are given below [see Algorithms 4-5 and 4-6]. All the user has to do is specify the positions of the two edges of the potential sheet, and the voltage of the potential sheet.

This process is very user-proof. The sadd_psheet() subroutine makes sure that the user doesn't try to put the first edge of the potential sheet outside of the problem region, or on a boundary. It doesn't let the user put it on top of a potential rod, or on top of a line charge. It does let him connect it to another potential sheet though. However, sadd_psheet() is careful to ensure that this new potential sheet will have the same voltage as the potential sheet that it's touching.

The subroutine spart2_add_psheet() does a lot more user-proofing. It makes sure that, if the first edge of the potential sheet touches an existing potential sheet, that the second endpoint does not touch a potential sheet of a different voltage, and that the new potential sheet does not intersect a potential sheet of a different voltage.

spart2_add_psheet() also checks to make sure that the second edge isn't at the same (x,y) point as the first edge. "Poisson's" number-crunching algorithm must have the potential sheets be horizontal, vertical, or at a 45-degree angle. spart2_add_psheet() makes sure that this is the case. It also makes sure that the user enters a numeric value for the voltage of the potential sheet, and that this value is within the allowable range. If either edge of the new potential sheet touches another potential sheet, or if the new potential sheet intersects another potential sheet, then spart2_add_psheet() does NOT ask the user for the voltage of the new potential sheet, because the new potential sheet must take on the same voltage as any potential sheet that it is touching.

If any of these tests fail, then either sadd_psheet() or spart2_add_psheet(...) tell the user what the problem is, and give the user the opportunity to correct the problem.

ALGORITHM 4-6: ADDING A POTENTIAL SHEET: PART1.

```
[1] User selects the "ADD potential rod" menu item.
        -Set the "connection_made" variable equal to NO.
         Set the "existing_voltage" variable equal to 0.0.
         Draw cross-hair in center of problem region.
         Tell user "Position edge of potential sheet."
         Tell user "NOTE:  Potential sheets can only be horizontal,
                    vertical, and at a 45-degree angle."
         Tell user "Press space bar to enter potential sheet; press
                    delete key to abort."

[2] Did user hit the delete key?
        -if YES, Erase cross-hair.
                 Tell user "Aborted -- add potential sheet."
                 Return to menu.

        -if NO,  Continue.

[3] Did user hit the ↑, ↓, ←, or → key?
        -if YES, Check to see if this would move the cursor out of the
                 problem region.

                 -if YES, Do nothing.

                 -if NO,  Move the cross-hair one grid unit in the
                          indicated direction.

        -if NO,  Continue.

[4] Did user hit the space bar?
        -if NO,  Go to [2].

        -if YES, [4a] Check the potential rod array:  does a potential
                      rod exist at this (x,y) grid point?

                      -if YES, Tell user "Rod of constant potential is
                                          here.  Try again."
                               Go to [2].

                      -if NO,  Continue.

                 [4b] Check the line charge array:  does a line charge
                      already exist at this (x,y) grid point?

                      -if YES, Tell user "Line charge is here.  Try
                                          again."
                               Go to [2].

                      -if NO,  Continue.

                 [4d] Turn on one pixel, to mark this edge of the sheet.
                      Save the (x,y) coordinates of this point.
                      Check potential sheet array:  is there already a
                      potential sheet at this point?
        -if YES, Make the "connection_made" variable
                 equal YES.
                 Make the "existing_voltage" variable be the
                 voltage of the potential sheet that was
                 here first.
                 Tell user "This new sheet of constant
                            potential will also have voltage
                            = X", where X is the voltage of
                            the first potential sheet.
                 Tell user "Position other edge."
                 Go to PART2, and pass it the values of
                 the following:  1)the coordinates of the
                                   first edge of the sheet.
                                 2)the "connection_made"
                                   variable.
                                 3)the "existing_voltage"
                                   variable.

        -if NO,  Tell user "Position other edge."
                 Go to PART2, and pass it the values of
                 the following:  1)the coordinates of the
                                   first edge of the sheet.
                                 2)the "connection_made"
                                   variable.
                                 3)the "existing_voltage"
                                   variable.
```

ALGORITHM 4-6: ADDING A POTENTIAL SHEET: PART2.

```
[5] Did user hit the delete key?
        -if YES, Erase cross-hair.
                 Tell user "Aborted -- add potential sheet."
                 Return to menu.

        -if NO,  Continue.

[6] Did user hit the ↑, ↓ , ← , or → key?
        -if YES, Check to see if this would move the cursor out of the
                 problem region.

                 -if YES, Do nothing.

                 -if NO,  Move the cross-hair one grid unit in the
                              indicated direction.

        -if NO,  Continue.

[7] Did user hit the space bar?
        -if NO,  Go to [5].

        -if YES, [7a] If the first endpoint of the new potential sheet
                      touches a potential sheet, check:  (1)does second
                      endpoint of new potential sheet touch a potential
                      sheet of a different voltage, or does (2)the new
                      potential sheet intersect a potential sheet of a
                      different value?

                      -if YES, Tell user "Potential sheets of 2 diff.
                                          values can't touch.  Try again."
                               Go to [5].

                      -if NO,  Continue.

                 [7b] Check the potential rod array:  does a potential
                      rod exist at this (x,y) grid point?

                      -if YES, Tell user "Rod of constant potential is
                                          here.  Try again."
                               Go to [5].

                      -if NO,  Continue.

                 [7c] Check the line charge array:  does a line charge
                      already exist at this (x,y) grid point?

                      -if YES, Tell user "Line charge is here.  Try
                                          again."
                               Go to [5].

                      -if NO,  Continue.

                 [7d] Check:  are the (x,y) coordinates of this point the
```

```
             same as the (x,y) coordinates of the first endpoint
             of the potential sheet?

             -if YES, Tell user "Second edge of sheet can't be
                                 same as first edge.  Try again."
                      Go to [5].

             -if NO,  Continue.

        [7e] Check, using the coordinates of the two endpoints
             of the sheet:  is the sheet horizontal, vertical,
             or at a 45-degree angle?

             -if YES, Continue.

             -if NO,  Tell user "Potential sheet must horiz.,
                                 vert., or at an angle of 45
                                 degrees.  Try again."
                      Go to [5].

        [7f] Draw a line from the first endpoint to the second
             endpoint.
             Check:  is "voltage_already_assigned" equal to YES?

             -if YES, Tell user "Sheet of constant potential has
                                 voltage = X", where X is the
                                 value of the variable
                                 "existing_voltage".
                      Return to menu.

             -if NO,  Continue.

        [7g] Tell user "Enter the voltage for sheet of constant
                        potential-->".
             Read in user's input.
             Check:  did user enter non-numeric data?

             -if YES, Tell user "Non-numeric data entered.  Try
                                 again."
                      Go to [7g].

             -if NO,  Continue.


        [7h] Check:  is the number that the user entered in the
             range of -10.0 <= voltage <= 10.0?

             -if YES, Tell user "Potential rod has voltage = X",
                      where X is the value the user entered.
                      For all the potential sheet array elements
                      between the endpoints of the potential
                      sheet, inclusive, make the "chosen" flag
                      equal YES, and make the "value" variable
                      equal X.
                      Make the "an_endpt" flag in the potential
                      sheet array be YES for the endpoints of
```

```
                              the potential sheet.
                              Put the coordinates of the endpoints of the
                              potential sheet and the voltage X into the
                              potential sheet database, and make the
                              "valid" flag equal YES.
                              Return to menu.

                    -if NO,   Tell user "Valid range:  -10.0 <= voltage
                              <= 10.0  Try again."
                              Go to [7g].
```

Deleting a potential sheet.

Deleting a potential sheet is done via the sdelete_psheet() subroutine [see Appendix K, file "fneditor.c", pages 51-53] and the spart2_delete_psheet(...) subroutine [see Appendix K, file "fneditor.c", pages 53-54]. The algorithms for both of these subroutines are given below [see Algorithm 4-7 and 4-8]. All the user has to do is specify the where the two edges of the potential sheet are.

As with everything else about the graphics editor, this process is very user-proof. The first thing that the sdelete_psheet() subroutine does is make sure that there are any potential sheets to delete. It then makes sure that the user doesn't try to delete a line charge or a potential rod instead. It also makes sure that the user specifies one of the two edges of the potential sheet; deleting PART of a potential sheet is not allowed!!

spart2_delete_psheet(...) makes sure that the user doesn't pick a line charge or a potential rod instead of the second edge of the sheet. It also makes sure that this is the edge of the potential sheet, not somewhere in the middle of it. It also checks to make sure that this is the second edge of the same potential sheet!

ALGORITHM 4-7: DELETING A POTENTIAL SHEET: PART1.

```
[1] User selects the "DELETE potential sheet" menu item.

[2] Check potential sheet array:  do any potential sheets exist?
        -if YES, Draw cross-hair in center of problem region.
                 Tell user "Move to edge of sheet."
                 Tell user "Press space bar to delete potential sheet;
                            press delete key to abort."

        -if NO,  Tell user "No potential sheets exist."
                 Return to menu.

[3] Did user hit the delete key?
        -if YES, Erase cross-hair.
                 Tell user "Aborted -- delete potential sheet."
                 Return to menu.

        -if NO,  Continue.

[4] Did user hit the ↑, ↓ , ←, or → key?
        -if YES, Check to see if this would move the cursor out of the
                 problem region.

                 -if YES, Do nothing.

                 -if NO,  Move the cross-hair one grid unit in the
                           indicated direction.

        -if NO,  Continue.

[5] Did user hit the space bar?
        -if NO,  Go to [3].

        -if YES, [5a] Check the potential rod array:  does a potential
                      rod exist at this (x,y) grid point?

                      -if YES, Tell user "Rod of constant potential is
                                          here.  Try again."
                               Go to [3].

                      -if NO,  Continue.

                 [5b] Check the line charge array:  does a line charge
                      already exist at this (x,y) grid point?

                      -if YES, Tell user "Line charge is here.  Try
                                          again."
                               Go to [3].

                      -if NO,  Continue.

                 [5c] Check the potential sheet array:  is this point part
                      of a potential sheet?
        -if YES, Continue.

        -if NO,  Tell user "No sheet of constant potential
                            exists here.  Try again."


    [5d] Check potential sheet database:  is this the
         endpoint of a potential sheet?

        -if YES, Note the (x,y) coordinates of this point.
                 Tell user "Move to other edge."
                 Go to PART2 and pass it the (x,y)
                 coordinates of the first endpoint.

        -if NO,  Tell user "This isn't the endpoint of a
                            a potential sheet.  Try again."
                 Go to [3].
```

ALGORITHM 4-6: DELETING A POTENTIAL SHEET: PART2.

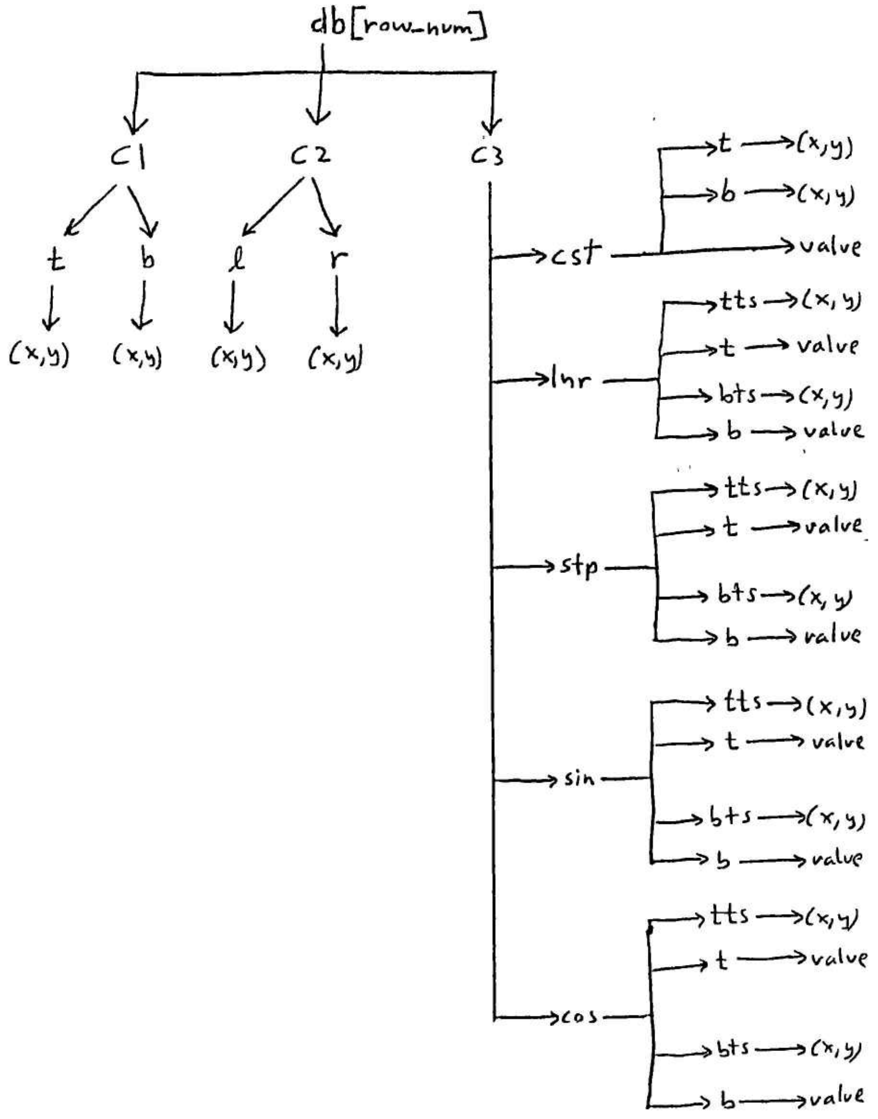
```
[6] Did user hit the delete key?
        -if YES, Erase cross-hair.
                 Tell user "Aborted -- delete potential sheet."
                 Return to menu.

        -if NO,  Continue.

[7] Did user hit the ↑, ↓ , ←, or → key?
        -if YES, Check to see if this would move the cursor out of the
                 problem region.

                 -if YES, Do nothing.

                 -if NO,  Move the cross-hair one grid unit in the
                              indicated direction.

        -if NO,  Continue.

[8] Did user hit the space bar?
        -if NO,  Go to [6].

        -if YES, [8a] Check the potential rod array:  does a potential
                      rod exist at this (x,y) grid point?

                      -if YES, Tell user "Rod of constant potential is
                                          here.  Try again."
                               Go to [6].

                      -if NO,  Continue.

                 [8b] Check the line charge array:  does a line charge
                      already exist at this (x,y) grid point?

                      -if YES, Tell user "Line charge is here.  Try
                                          again."
                               Go to [6].

                      -if NO,  Continue.

                 [8c] Check the potential sheet array:  is this point
                      part of a potential sheet?

                      -if YES, Continue.

                      -if NO,  Go to [6].

                 [8d] Check:  are the (x,y) coordinates of this endpoint
                      the same as the (x,y) coordinates of the first
                      endpoint?

                      -if YES, Tell user "Second edge of sheet can't be
                                          same as first edge.  Try again."
```

```
                Go to [6].

        -if NO,  Continue.

[8e] Check the potential sheet database:  are these two
     potential sheet endpoints the endpoints of the SAME
     potential sheet?

     -if YES, Continue.

     -if NO,  Go to [6].

[8f] Erase line.
     Invalidate all the points which made up this
     potential sheet, in the potential sheet array, by
     making the "chosen" flag equal NO.
     Invalidate this potential sheet in the potential
     sheet database, by making the "valid" flag equal NO.
     Tell user "Sheet of constant potential has been
                deleted."
     Return to menu.
```

4.6 EXPLANATION OF DATA STRUCTURES USED BY USER INTERFACE.

The following in an explanation of the organization of the various structures and arrays used by the user-interface. Their use is explained in the preceeding sections of this chapter. These data structures are defined in Appendix K, files "structures.c" and "defs".

The structure of the text screen coordinate database



Key to variable nomenclature:

variable	meaning
row_num	This indicates one of the walls of the problem box; 0, 1, 2, and 3 indicate the top, left, right, and bottom walls, respectively,
db	database
C1, C2, C3	Columns 1, 2, and 3 of the text grid coordinate system [see Figure 4-2].
cst lnr stp sin cos	constant linear step sine cosine
t, l b, r	top, left bottom, right
tts bts	top text string bottom text string
(x,y)	a pair of integers, to indicate a position on the screen, in terms of rows and columns. Screen is 24 rows by 80 columns.
valve	A double precision floating point; holds the value of the numerical parameter of a boundary function

The function of db[] is to hold the (x,y) screen coordinates for drawing text in each of the 4 rows and 3 columns of the main screen [see Figure 4-3], no matter what type of boundary condition is being displayed. db[] also holds all of the values associated with the numerical parameters for the boundary conditions for all four of the problem box's walls.

EXPLANATION OF THE "options[][]" DATABASE

```
Declaration:  static int options[4][3];


Indexing:     options[R1][C1]  [R1][C2]  [R1][C3]
                     [R2][C1]  [R2][C2]  [R2][C3]
                     [R3][C1]  [R3][C2]  [R3][C3]
                     [R4][C1]  [R4][C2]  [R4][C3]
```

---This is a two-dimensional array of integers. It ia used by the main screen, to keep track of what options are selected for each of the four boundaries of the problem box. The integers are used as flags (whose values may be OPT1, OPT2, OPT3, OPT4, or OPT5) to indicate which of the options is being displayed [see Figure 4-1].

---R1, R2, R3, R4, C1, C2, and C3 are constants, and are defined in the "defs" file [see Appendix H]. R1 implies "row 1", and C1 implies "column 1".

---The following in a table showing which options are allowed in which columns [see Figure 4-1], and what the option number is that is associated with each option. OPT1, OPT2, OPT3, OPT4 and OPT5 are constants, and are defined in the "defs" file [see Appendix H].

For column C1:

option number	what option is displayed
OPT1	potential
OPT2	normal derivative

For columns C2 and C3 (these two columns must always be displaying the same option):

option number	what option is displayed
OPT1	constant
OPT2	linear
OPT3	step
OPT4	sine
OPT5	cosine

EXPLANATION OF THE "LINE CHARGE ARRAY"

It is a grid array, called lc_grid[][].

Declaration: static struct grid_point lc_grid[33][33];

---This is a two-dimensional array of structures. It is used by the graphics editor to keep track of the line charges. There are 33x33 points in the problem region, therefore the line charge array is 33x33. It is defined in Appendix H, in the file "structures.c".

---Each element of the array has the following information:

```
        double x, y;  ------the VT240 terminal coordinates (range:
                            x is 0 to 799, y is 0 to 470) of the
                            pixel at this grid point.

        double value;  -----the lambda associated with the line charge
                            at this grid point.

        int chosen;  -------a flag (YES or NO), to indicate if there is
                            a line charge at this grid point or not.

        int num;  ----------an index; the first line charge is given an
                            index of 1, the next is called 2, etc.
```

EXPLANATION OF THE "POTENTIAL ROD ARRAY"

It is a grid array, called pr_grid[][].

Declaration: static struct grid_point pr_grid[33][33];

---This is a two-dimensional array of structures. It is used by the graphics editor to keep track of the potential rods. There are 33x33 points in the problem region, therefore the potential rod array is 33x33. It is defined in Appendix H, in the file "structures.c".

---Each element of the array has the following information:

```
        double x, y;  -------the VT240 terminal coordinates (range:
                             x is 0 to 799, y is 0 to 470) of the
                             pixel at this grid point.

        double value;  ------the voltage associated with the potential
                             rod at this grid point.

        int chosen;  --------a flag (YES or NO), to indicate if there
                             a potential rod at this grid point or not.

        int num;  -----------an index; the first potential rod is given
                             an index of 1, the next is called 2, etc.
```

EXPLANATION OF THE "POTENTIAL SHEET ARRAY"

It is a grid array, called ps_grid[][].

Declaration: static struct psheet_grid_point ps_grid[33][33];

---This is a two-dimensional array of structures. It is used by the graphics editor, to keep track of all the points in each potential sheet. Even though a potential sheet is nothing but a series of potential rods, it is necessary to keep a separate array, so that "Poisson" can tell the difference between a potential sheet and two (or more) potential rods in a row. There are 33x33 points in the problem region, therefore the potential sheet array is 33x33.

---Each element of the array has the following information:

```
double x, y;  ------the VT240 terminal coordinates (range:
                    x is 0 to 799, y is 0 to 470) of the
                    pixel at this grid point.

double value;  -----the voltage associated with the potential
                    sheet which goes through this grid point.

int chosen;  -------a flag (YES or NO), to indicate if a
                    potential sheet passes through this grid
                    point or not.

int num;  ----------an index; the first potential sheet is
                    given an index of 1, the next is called
                    2, etc.

int an_endpt -------a flag (YES or NO), to indicate if this
                    is an endpoint of a potential sheet (as
                    contrasted to a point somewhere between
                    the two endpoints of the line marking
                    the position of the potential sheet).
```

EXPLANATION OF THE DATABASE "ps_db[]"

Declaration: static struct psheet_point ps_db[MAX_SHEETS];

---This is a one-dimensional array of structures. It is used by the graphics editor to keep track of the endpoints of each potential sheet. It is defined in Appendix H, in the file "structures.c".

---Each element of the array has the following information:

int p0.xci, p0.yci	--------the x and y "current index" for the position of the first edge of the potential sheet.
int p1.xci, p1.yci	--------the x and y "current index" for the position of the second edge of the potential sheet.
double value	--------------the voltage of this potential sheet.
int num	-------------------an index; the first potential sheet is given an index of 1, the second is called 2, etc.
int valid	-----------------a flag (YES or NO), to indicate whether or not this potential sheet has been deleted, or if it's still valid.

Chapter 5
Future improvements to enhance "Poisson".

5.1 Terminal independence.

Because ReGIS is terminal-dependent, and because Penplot is NOT truly terminal-independent, "Poisson" will only run on certain terminals [see Appendix C]. In the future, there is supposed to be an industry-wide terminal-independent UNIX graphics system: the Graphics Kernal System (GKS). If and when such a system comes into being, it would be worth the effort to modify "Poisson" to use GKS, because this would exponentially increase the number of users who could have access to "Poisson".

5.2 Use of a mouse.

Presently line charges, rods of constant potential (both of which look like dots, when mapped from three dimensions into two dimensions), and sheets of constant potential (which look like lines, when mapped from three dimensions into two dimensions) are positioned inside the problem region with the use of four arrow keys (up, down, left, and right) [see Figure 2-9]. The terminal that "Poisson" works on now (a DEC VT240) does not have a mouse; that's why the arrow keys are used. If "Poisson" ever gets ported onto a terminal which DOES have a mouse, then the person doing the porting should modify "Poisson" so that the user can use the mouse instead of (or as an option to) the arrow keys. A mouse would be a faster and easier way for the user to enter line charges, rods of constant potential, and sheets of constant potential.

5.3 A neat way to add more menu items: a "paged" menu.

Right now the two menus used in "Poisson" are of the form below:

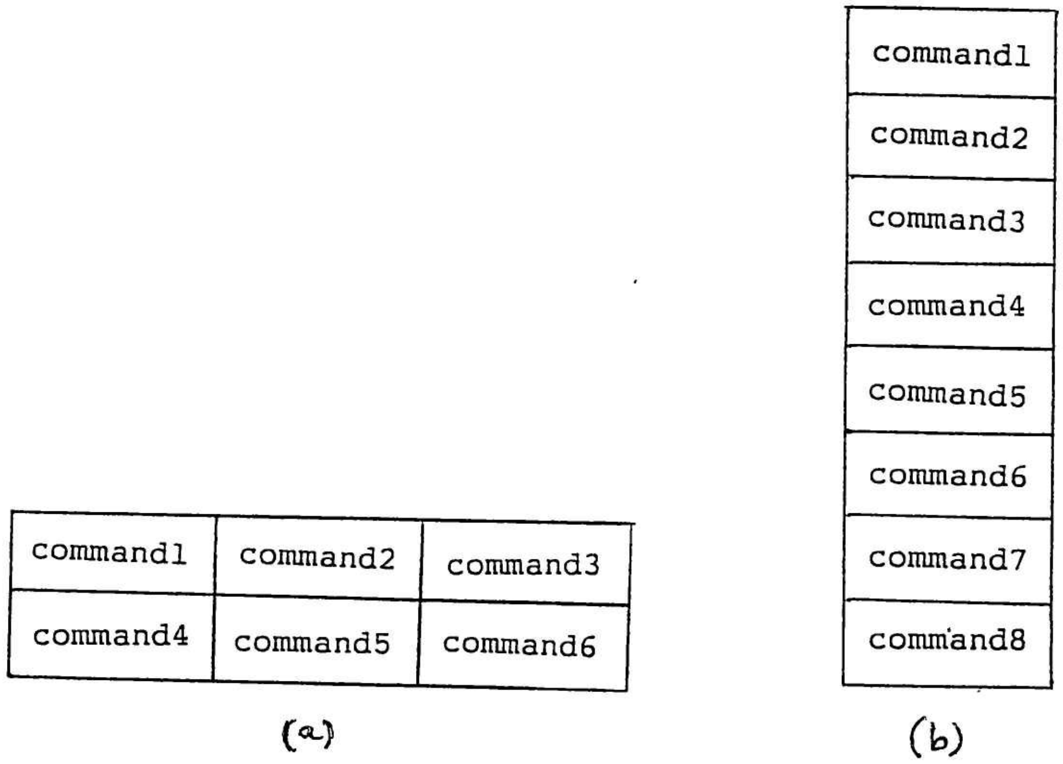


Figure 5-1: The menus currently used in "Poisson".

For the main screen, menu (a) is used; for the graphics editor screen, menu (b) is used.

When further work is done on this program, or if it becomes desireable to expand the command set for some other reason, it is NOT necessary to enlarge the size of the menus! You wouldn't want to expand the size of the menu (to fit more text into each menu slot) because that would take away more screen space, and necessitate MAJOR amounts of code re-writing (because everything else on the screen would have to be somehow condensed).

There are two ways to add more commands without making the menu larger. The first way is a brute force approach; the second way is much more desireable. The brute force method for adding more commands would be to change a menu which looks like this:

cmd1	cmd2	cmd3
cmd4	cmd5	cmd6

Figure 5-2: The current menu.

to one like this:

cmd1	cmd2	cmd3	cmd4
cmd5	cmd6	cmd7	cmd8

i.e., reduce the space for each command in order to add in more commands.

Figure 5-3: A bad way to expand the command selection.

This has the drawback that it forces the command names (i.e., the text in the menu slots) to become more concise, which makes the commands more cryptic, less intuitive, more ambiguous, and more confusing!

Fortunately there is a very elegant and simple solution: have more than one <u>page</u> to the menu! Turning the pages is simply a matter of selecting the "next page" menu item. When the user selects the next page, the old menu's commands will be erased, and the new menu's commands will be written in their place. This way it is possible to have an infinite number of commands, and never have to increase the size of the menu! See Figure 5-4.

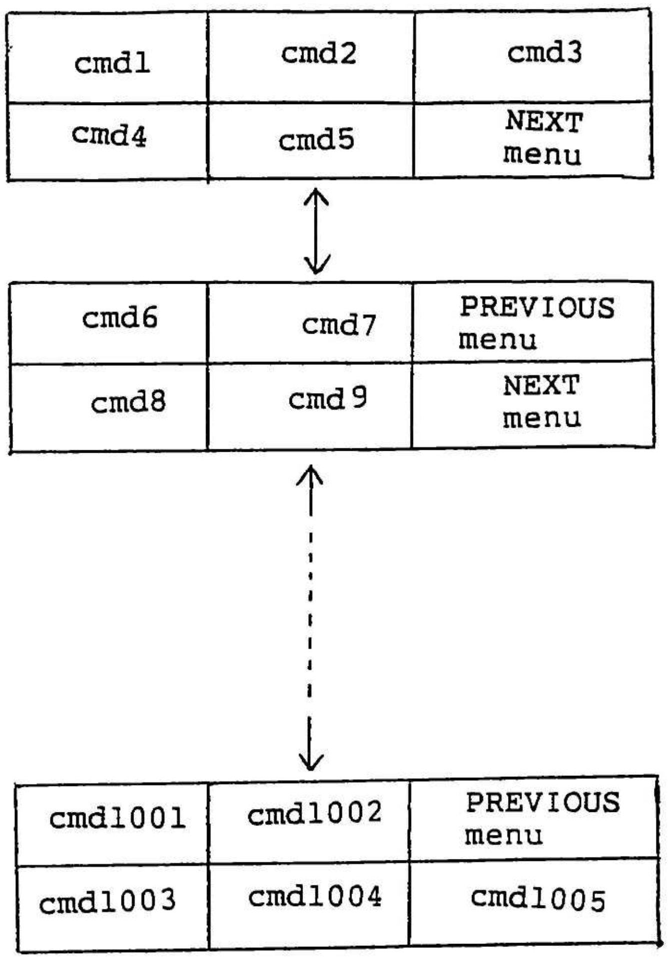


Figure 5-4: A paged menu scheme.

5.4 Labeling things.

It would be nice if, in the future, things on the graphics screen were labelled. For example, now the graphics editor screen looks like:

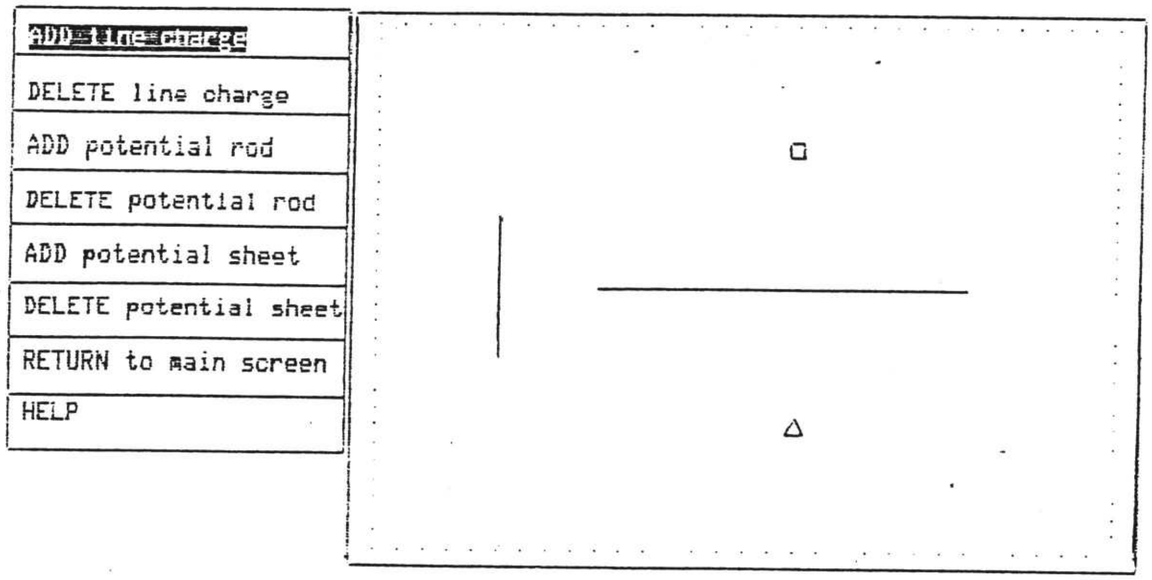


Figure 5-5: The present graphics editor screen.

It would be nice if it were instead:

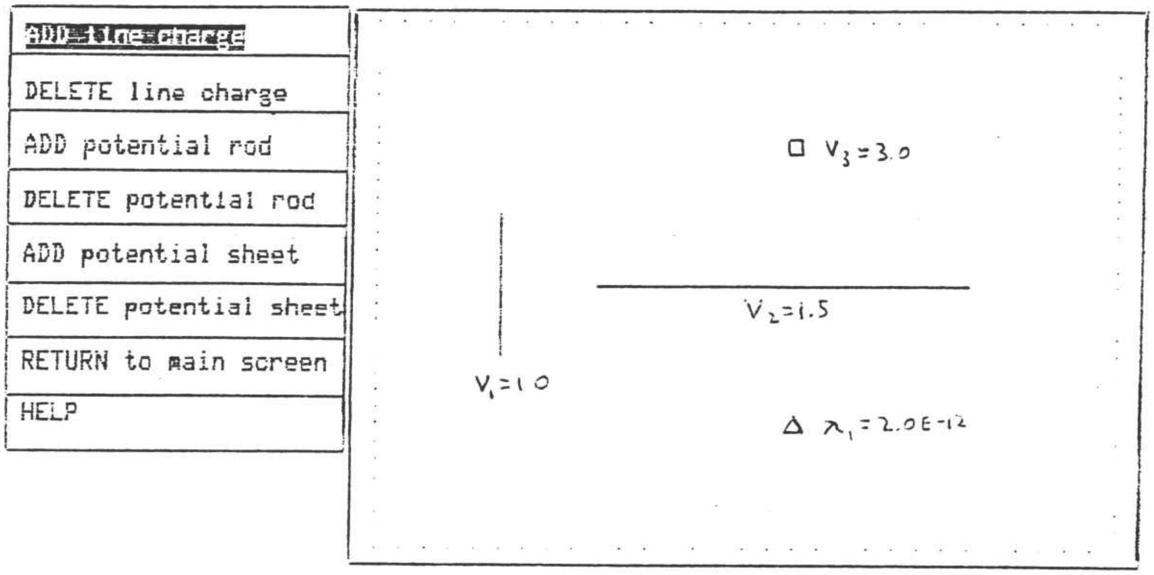


Figure 5-6: The new and improved graphics editor screen.

This SEEMS like a fairly straightforward matter, but it is actually a very difficult problem to solve. The main difficulty is that when there are a lot of things (i.e., line charges, rods of constant potential, and sheets of constant potential) on the screen, the labels may overlap and become confusing or unreadable.

A first step towards solving this problem is to write the labels in as small a font as the terminal will support.

A further improvement would be to put just an index number by each object, and then provide a table somewhere else on the screen, with a listing of all the index numbers and the values associated with them.

5.5 Anti-aliasing, i.e., making the contours smoother.

Anti-aliasing, i.e., making the contour lines less jagged, is another desirable feature which ought to be investigated in the future.

The following excerpt from a well-known book on computer graphics[3] explains the concept of anti-aliasing very well.

To make use of these anti-aliasing techniques requires a graphics video terminal which has a "grey scale". A grey scale means that a pixel is not merely either on (white) or off (black); rather, it can have several different levels of brightness (different shades of grey). For instance, with a 2-bit grey scale, each pixel can have 2 different values: 1/4 on (1/4 bright), 1/2 on (half bright), 3/4 on (3/4 bright) and full brightness. The DEC VAXstation 100 workstations do NOT have a grey scale.

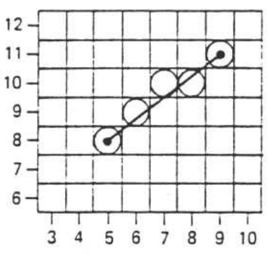


Fig. 11.3 Line from point (5, 8) to point (9, 11) drawn with Bresenham's algorithm.

The line appears jagged, in part because of the enlarged scale of the drawing and in part due to the approximations involved in attempting to draw a line on a discrete grid of points.

11.2.3 Antialiasing Lines

A more pleasing line can be drawn by applying what have come to be known as *antialiasing* and *dejagging* techniques. These techniques, which have their roots in sampling theory, were first applied to graphics by Catmull [CATM74, CATM78a], Crow [CROW77b], and Shoup [SHOU73]. The essential idea is that a pixel, which has a nonzero area on the screen, should be used to represent the nonzero area of the world which is mapped onto the pixel, as depicted in Fig. 11.4. A necessary corollary is that visible lines and characters in the real world have nonzero width; they are no longer mathematical entities made up of line segments of zero width.

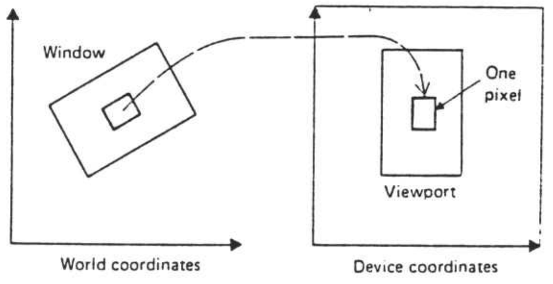


Fig. 11.4 Rectangular area in world coordinates maps into the area covered by one pixel on the screen.

How can we apply this notion? Figure 11.5 shows a line of nonzero width superposed on a raster. The raster grid has been shifted in x and y by half a unit, because we want to focus on the area covered by the pixels which are now positioned in the *center* of each grid box, not on the grid intersections. Thus, a pixel is represented by a square area within the grid. (This is, in itself, an idealization: the intensity distribution of an intensified pixel is approximately normal, and the tails of the distribution overlap into adjacent pixels.)

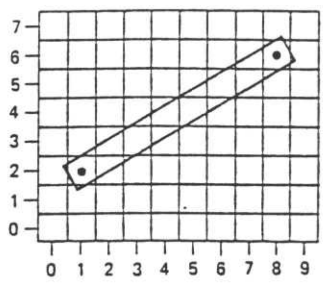


Fig. 11.5 Line of nonzero width from point (1, 2) to point (8, 6).

Each pixel overlapped by the line must have an intensity proportional to the area of the pixel covered by the line. Thus for a white line on a black background, pixel (2,2) would be about 50% white while pixel (3,2) would be about 10% white. Pixels such as (2,4) would be completely black. (For lines of less than maximum intensity, these percentages would be scaled down accordingly.) Figure 11.6 shows lines drawn with and without this type of antialiasing. Note that the smoothing of the lines is achieved at the expense of a slight blurring of the line edges.

Computing the fraction of each pixel overlapped by the rectangular area of the line can be quite time-consuming. Crow [CROW78b], and Barres and Fuchs [BARR79] have developed relatively efficient ways to organize the computations, but the latter's algorithm requires that all line segments be specified before any pixels are generated. Speed will increase in the future (Piller [PILL80] and Gupta et al [GUPT81b] have developed hardware-implemented parallel processing approaches) but, if speed is most important, alternatives are either to live with jagged lines or to use a larger refresh buffer with a high-resolution CRT. Doubling the refresh buffer in both x and y from the typical 512 to 1024 quadruples the number of pixels, doubles the time to scan-convert a line into the buffer, and does not completely remove jagged edges. However, scan-converting a line at doubled resolution is typically faster than antialiasing the line at the original resolution.

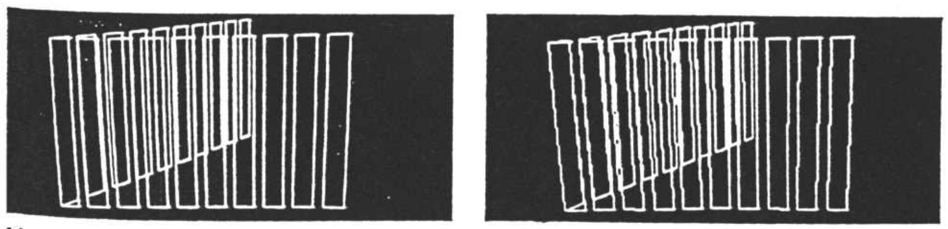


Fig. 11.6 Lines displayed (a) with and (b) without antialiasing (courtesy Jose Barros and Henry Fuchs).

11.2.4 Constant-Intensity Lines

Antialiasing solves another problem encountered with the more straightforward approaches. Consider the two scan-converted lines shown in Fig. 11.7. The diagonal line has a slope of 1, and hence is $\sqrt{2}$ times longer than the horizontal line. Yet each line has the same number of pixels (10) set on. If the intensity of each pixel is I, then the intensity per unit length of line A is I, while for line B, it is only $I/\sqrt{2}$, which can be easily detected by a viewer. A simple way to correct this particular problem is to compensate the intensity used to display a line to account for the line's slope. Antialiasing, however, achieves this same objective because the area covered by the line determines how much intensity is distributed along its length. If we consider the lines in Fig. 11.7 as rectangles of height w, then line A would cover $10w$ units of area and line B would cover $10\sqrt{2}w$ units, and intensity per unit length will be constant.

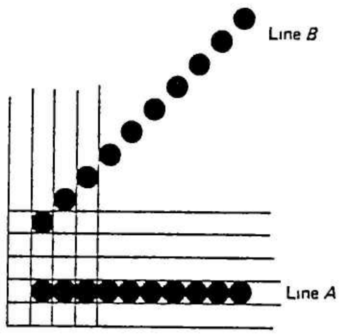


Fig. 11.7 The unequal-intensity problem.

REFERENCES

[1] DEC VT125 Graphics Terminal User Guide,
number EK-VT125-U6-002

[2] Penplot Graphics Control System Reference Manual,
MIT Joint Computer Facility, 1981

[3] James D. Foley and Andries Van Dam,
Fundamentals of Interactive Computer Graphics,
pp436-438, Addison-Wesley Publishing Company,
Menlo Park, CA, 1982

Appendix A

An introduction to the "Poisson" program.

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

Department of Electrical Engineering and Computer Science

6.013 – Electromagnetic Fields and Energy

October 1986

Project ATHENA POISSON PROGRAM

Version 1.0

Introduction

The program Poisson calculates and gives a contour plot of the potential and plots the electric field lines* on a square grid of 33×33 points (32×32 spaces) or 65×65 points (64×64 spaces) with the following possible specifications:

1. Boundary conditions:

   a) Dirichlet (i.e., potential)

   b) Neumann (i.e., normal derivative of potential)

   c) Mixed Dirichlet and Neumann.

   The potential or normal derivative specified on any boundary may have any one of the following functional dependencies: 1. constant, 2. linear, 3. step, 4. sine, 5. cosine.

2. Potentials at any of 31×31 internal grid points.

3. Line charge density at any of 31×31 internal grid points.

The above specifications are entered via a menu. (The following section "Running the POISSON Program for the First Time," explains the use of the menu.)

Specified potentials should be in the range ± 1 Volt to ± 10 Volts. (See attached "Explanation of Error Criteria for the Poisson Program.")

Specified normal derivatives of potential (Neumann boundary conditions) should be in the range 10^{-2} to 10^{-1} (1e-2 to 1e-1) Volts/unit length. (See attached "What is a Reasonable Size for dV/dn?")

Specified line charge densities should be in the range 2×10^{-11} to 2×10^{-10} (2e-11 to 2e-10) Coulomb/unit length. (See attached "What is a 'Good Choice' for λ?")

The computational technique used is: finite-difference relaxation with SOR, including a few extra features to make the computation faster on the Athena VAX 750. (See attached "A Solution of Poisson's Equation by the Relaxation Method.")

* Note: At present we can only plot field lines if there are no internal charges or potentials specified.

Running the POISSON Program for the First Time

Note: At this time, the program can only be run on a terminal which has a VT240 graphics box connected to it. In the Set-Up you should set the XOFF option (on the Communications Screen) to "XOFF at 1024".

To run the program, log onto ATHENA. If your account is on the machines "hactar" or "aphrodite", then type "~bers/poisson" in response to the system prompt. If your account is on a different machine, type "rcp hactar: ~bers/poisson poisson". Then when you get the system prompt again, type: "poisson". The screen will display a brief description of the program. When you have finished reading, press "return". The main menu is then displayed and shows a box with the default values of potentials (in Volts) on its four boundaries (namely: 1, 0, 0, and 0). All possible commands are also located in the menu, which is at the bottom of the screen.

To move the cursor, for either changing the default boundary values or to invoke the commands, use the 4 arrow keys:

↑	move up
↓	move down
→	move right
←	move left

You can change any or all of the assigned default boundary values. To change the boundary conditions on a side from a specified "POTENTIAL" (Dirichlet condition) to specified "NORMAL DERIVATIVE" (Neumann condition), or vice-versa, simply toggle back and forth by pressing the "space bar". The functional dependency of the potential or normal derivative on a given boundary is selected by moving the cursor to the field whose default value is "constant" and pressing the space bar until the desired function appears on the screen. Once the type of function is selected, the user is prompted to supply the necessary parameters as follows:

1. Constant:	user specifies value of potential or normal derivative.
2. Linear:	user specifies the values at the endpoints of the boundary.
3. Step:	user specifies two values, one for each half of the boundary.
4. Sine:	a * sin (2 * pi * b * x), with $0 < x < 1$, on "top" (y= 1) or "bottom" (y= 0) boundaries; a * sin (2 * pi * b * y), with $0 < y < 1$, on "left" (x= 0) or "right" (x= 1) boundaries. User specifies the values a and b.
5. Cosine:	a * cos (2 * pi * b * x), with $0 < x < 1$, on "top" (y= 1) or "bottom" (y= 0) boundaries; a * cos (2 * pi * b * y), with $0 < y < 1$, on "left" (x= 0) or "right" (x= 1) boundaries. User specifies the values a and b.

These parameters are specified by moving the cursor to the location of the default values, hitting the "space bar", then entering the desired value, and pressing "return".

The commands shown at the bottom of the main menu screen are:

- ADD CHARGES OR POTENTIALS: to enter equipotential rods and/or line charges within the region
- SOLVE FOR POTENTIALS: to calculate the potential
- QUIT PROGRAM: to exit the program
- HELP: to get help
- RECALL SOLVED PROBLEM: to recall a problem previously solved and saved
- CHANGE GRID SIZE 32×32 or 64×64: to change grid size in which the program is calculated

To invoke any one of the commands, move the cursor to the location of the desired command (the command will then be highlighted) and press the "space bar" ($< sp >$).

To see how all of this is used it is best to go through an example or two. The rest of this writeup leads you through each step in setting up, solving and displaying the results of a problem, and saving and recalling it if you wish.

Let's do a sample problem. Suppose we want to calculate the potential and field lines between two parallel planes on which the potential is imposed to vary sinusoidally with x. Let the planes be located at $y = 0$ and $y = 1$, and let the potential on the planes vary as $\sin 2\pi x$. Let the grid size be the default value 32×32.

Enter	**Computer Response on Screen**
$\rightarrow$	The highlighted region moves one position right.
$< sp >$	The functional dependency on the top boundary changes to a linear function.
$< sp >$	The functional dependency changes to a step function.
$< sp >$	Sine function on the top boundary is chosen.
$\downarrow$	The highlighted region moves one position down.
$\downarrow$	The highlighted region moves one position down.
$\downarrow$	The highlighted region moves one position down.
$< sp >$	The functional dependency on the bottom boundary changes to a linear function.
$< sp >$	The functional dependency changes to a step function.
$< sp >$	Sine function on the bottom boundary is chosen.
$\downarrow$	The highlighted region moves one position down.
$< sp >$	Selects the "SOLVE FOR POTENTIAL" menu item. Main screen is erased. The program asks you to wait while it initializes array values. Then it asks if you want the array values displayed after each iteration. We don't, hence:
n $< cr >$	The program asks you to wait while it solves the problem. When it's done the boundaries of the region appear on the screen, and we are asked how many contours we want to see plotted in the existing range of potentials (in this case from -1 Volt to $+1$ Volt). Suppose we want 8 contours:
8 $< cr >$	Plots 8 contours (see Fig. 1 attached). To find the spacing between successive potentials simply divide the range of potentials (listed beneath plot) by $n + 1$, where n is the number of contours requested (in this case 8).

The program now asks if we would like to replot the potential contours, plot fields lines*, return to the main menu, save the solved problem in memory, or exit the program. If we wanted fewer or more contours: e.g., if you would like to see what happens when one of the contours includes the side boundary potential, i.e. zero Volts, type p $< cr >$ followed by e.g. 9 $< cr >$. On the other hand, suppose we would like electric field lines plotted over the contour plot of potential, then type:

* Note: At present we can only plot field lines if there are no internal charges or potentials specified.

Enter	Computer Response on Screen
f < *cr* >	We are asked how many field lines we would like to see plotted. The electric field at any point is a vector tangent to the field through that point and its magnitude is proportional to the density of field lines perpendicular to the field's direction. Suppose we want six field lines:
6 < *cr* >	Plots 6 field lines on top of the existing potential contours** (see Fig. 2). The program asks again if we want to replot potentials, plot field lines, return to the main screen or exit the program.

Now suppose we would like to see how the potential in this problem is modified by the addition of a rod placed at the center and held at 1 Volt. Hence we would like to modify the program; so type:

r < *cr* >	We return to the main screen. We would like to add a potential rod within the region. Type:
←	Highlighted region moves left.
< *sp* >	The "ADD charges or potentials" menu item is selected. Main screen disappears, and is replaced with the graphics editor screen.
↓	Highlighted region moves one position down.
↓	Highlighted region moves one position down.
< *sp* >	The "ADD potential rod" menu item is selected. A tiny graphics cursor will appear in the center of the graphics region of the screen, at $(i, j) = (16, 16)$. The (i, j) grid position of the cursor is displayed in the lower left-hand corner of the screen, beneath the menu.
< *sp* >	An equipotential rod is drawn at $(i, j) = (16, 16)$ (its symbol is a square). A prompt asks for a voltage value for this rod (rod #1).
1 < *cr* >	The rod is assigned a value of 1.0 Volts. The graphics cursor disappears, and you are returned to the graphics editor's menu.
↓	Highlighted region moves one position down.
↓	Highlighted region moves one position down.
↓	Highlighted region moves one position down.
↓	Highlighted region moves one position down.
< *sp* >	The "RETURN to main screen" menu item is selected. The main screen replaces the graphics editor screen. An equipotential rod can be seen in the problem region (it looks like a tiny dot).
→	The highlighted region moves right.

** Note that at the corners the accurate calculation of derivatives is difficult and the field lines many not be correct. To improve this try either choosing fewer field lines or a 64×64 grid with the same number of field lines.

Enter	Computer Response on Screen
< *sp* >	Selects the "SOLVE FOR POTENTIAL" menu item. Main screen is erased. The program asks you to wait while it initializes array values. Then it asks if you want the array values displayed after each iteration. We don't, hence:
n < *cr* >	The program asks you to wait while it solves the problem. When it's done the boundaries of the region appear on the screen, and we are asked how many contours we want to see plotted in the existing range of potentials (in this case from −1 Volt to +1 Volt). Suppose we want ten contours:
10 < *cr* >	Plots 10 contours (see Fig. 3).
	The program now asks if we would like to replot the potential contours, plot field lines, return to the main menu, save the solved program in memory,* or exit the program. Since there is a potential rod within the region we cannot (at this time) plot field lines. To start experimenting on your own type:
r < *cr* >	

Try changing the potentials on the boundaries, or changing the equipotential rods within the boundaries, or adding charged rods within the boundaries, and see what happens on the contour plot.

Once you have quit the program, type "logout < *cr* >" to leave the system.

*NOTE:

1. To save a solved program and its display type:

s < *cr* >	The program asks you to give a name to the problem. For example, to give it the name, "one" you type:
one < *cr* >	The program then returns to ask you to choose between p, f, r, and e, as before.

2. To retrieve a saved problem, return to the main menu, use arrow keys to highlight RECALL SOLVED PROBLEM, and press the "space bar". The program will then ask for the name of the solved problem; then, for example, type:

one < *cr* >	The program will then redisplay the main menu with the parameters of the saved problem. In order to see it displayed use the arrow key:
↑	This will highlight SOLVE FOR POTENTIALS, after which press
< *sp* >	The program will then directly display the solved problem.

3. To delete a saved problem, you must first exit the program (or invoke QUIT PROGRAM if you are in the main menu):

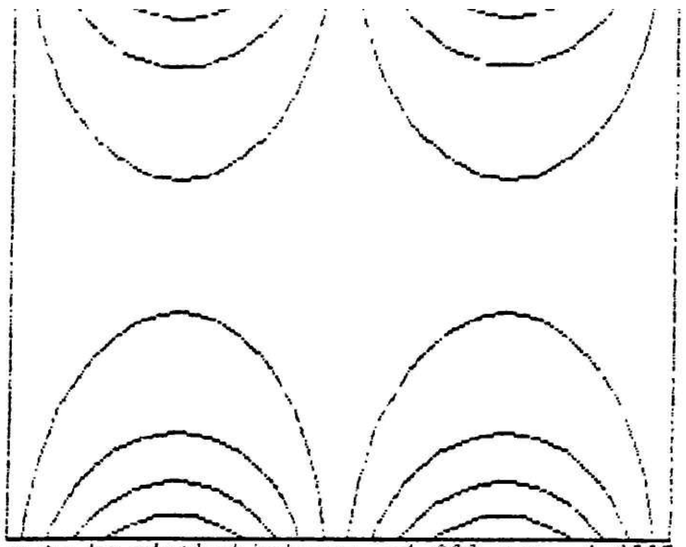
e < *cr* >	The system will display the system prompt, e.g., "hactar%", after which you type "rm" followed by the name of the problem to be erased, e.g.:
rm one < *cr* >	The system will then again display the system prompt ("hactar%"). To reenter the program, type:
poisson < *cr* >	

This program was developed by:

Denise Barnett	home: 494-1077
	denise@hactar
Ted Johnson	dorm line: 5-6432; outside line: 225-0628
	tcj@prill
G. Francis	3-2539
A. Bers	3-4195

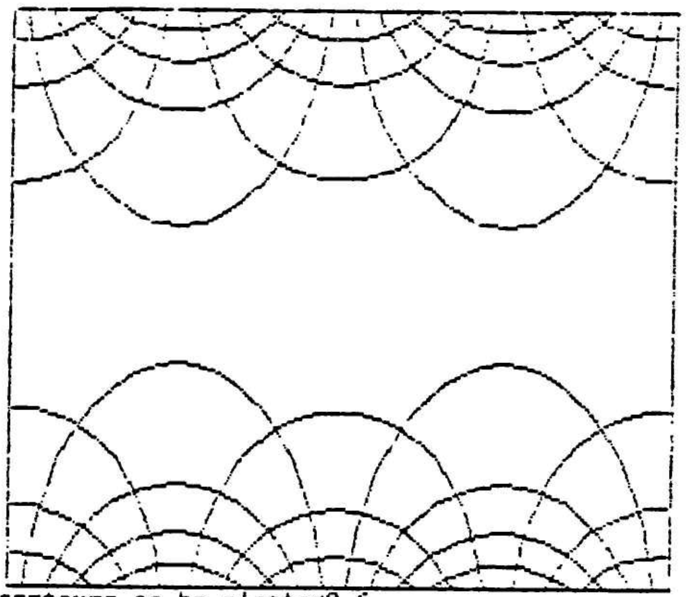
Feel free to call on any of us for help and/or with suggestions for improving the program.

Figure 1



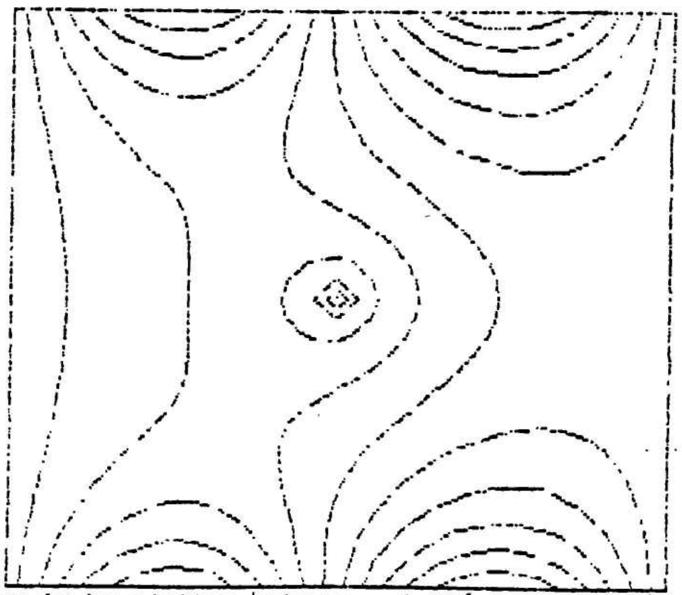
```
How many contours to be plotted between  -1.000 and   1.000? Enter p (replot pot
ential contours), f (plot field lines),
 r (return to main screen), s (save problem) or e (exit program):
```

Figure 2



```
How many field contours to be plotted? 6
Enter p (replot potential contours), f (plot field lines),
 r (return to main screen), s (save problem) or e (exit program):
```

Figure 3



```
How many contours to be plotted between  -1.000 and   1.000? 10
Enter p (replot potential contours), f (plot field lines),
 r (return to main screen), s (save problem) or e (exit program):
```

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

Department of Electrical Engineering and Computer Science

6.013 – Electromagnetic Fields and Energy

August 1986

Project ATHENA POISSON PROGRAM

D. Barnett, G. Francis, and A. Bers

ATTACHMENTS

Attachment 1 to POISSON Program

A SOLUTION OF POISSON'S EQUATION BY THE RELAXATION METHOD

Dirichlet Boundary Conditions

In rectangular coordinates Poisson's equation in *two dimensions* is of the form

$$\frac{\partial^2 V}{\partial x^2} + \frac{\partial^2 V}{\partial y^2} = -\frac{\rho(x,y)}{\epsilon_0} \tag{1}$$

We are given a square region where the potential is known on each boundary and we want to solve Poisson's equation on a grid of points within the region. There is also a line charge density $[\lambda(x,y)]$ given for each of the grid points. If the line charge density at each point is equal to 0 then the equation reduces to Laplace's equation:

$$\frac{\partial^2 V}{\partial x^2} + \frac{\partial^2 V}{\partial y^2} = 0\,.$$

The relaxation method is an approximate solution method but by iteration (repeated application of the method) the solution can be as accurate as desired.

The first term in (1) is the second derivative of V with respect to x, or the rate of change of the rate of change of V in the x direction. And the second term is likewise the rate of change of the rate of change of V in the y direction.

Consider the point P in the center of the given square region. Let the potential at P be represented by V_P and the line charge at P be represented by λ_P. Let the potential on the top boundary of the region be represented by V_t, on the right boundary by V_r, on the bottom by V_b, and on the left by V_ℓ (see Figure 1).

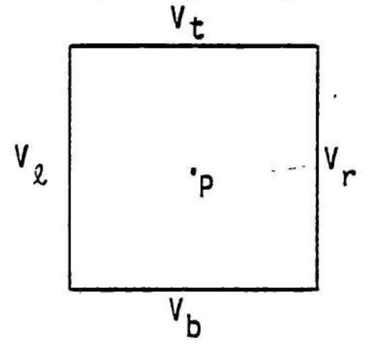


Figure 1

The distance from P to V_ℓ and from P to V_r is Δx, and the distance from P to V_t and from P to V_b is Δy. The slope of V between P and the right boundary is $(V_r - V_P)/\Delta x$. This is approximately equal to $\partial V/\partial x$. And the slope of V between P and the left boundary is

$(V_P - V_\ell)/\Delta x$. The difference between these two slopes divided by Δx is approximately equal to $\partial^2 V/\partial x^2$:

$$\frac{[(V_r - V_P)/\Delta x] - [(V_P - V_\ell)/\Delta x]}{\Delta x} \simeq \frac{\partial^2 V}{\partial x^2}$$

and similarly,

$$\frac{[(V_t - V_P)/\Delta y] - [(V_P - V_b)/\Delta y]}{\Delta x} \simeq \frac{\partial^2 V}{\partial y^2}$$

Therefore, substituting into equation (1) we obtain

$$\frac{[(V_r - V_P)/\Delta x] - [(V_P - V_\ell)/\Delta x]}{\Delta x} + \frac{[(V_t - V_P)/\Delta y] - [(V_P - V_b)/\Delta y]}{\Delta y} = -\frac{\rho_P}{\epsilon_0} \qquad (2)$$

For a square region, $\Delta x = \Delta y$, so equation (2) becomes

$$V_t + V_r + V_b + V_\ell + \frac{\lambda_P}{\epsilon_0} - 4V_P = 0$$

where $\rho_P(\Delta x)^2 = \lambda_P$ is the line charge density (charge per unit length in z) at P. Solving for V_P

$$V_P = \frac{1}{4}\left(V_t + V_r + V_b + V_\ell + \frac{\lambda_P}{\epsilon_0}\right) \qquad (3a)$$

Once again, if there is no line charge at point P (i.e., $\lambda_P = 0$),

$$V_P = \frac{1}{4}(V_t + V_r + V_b + V_\ell) \qquad (3b)$$

Thus we can find the potential at any point given the potential at four surrounding points and the line charge density at that point.

For the following discussion of this problem we will assume that the line charge density within the entire region is equal to zero, and we will work out the solution to Laplace's equation.

Initialization

To begin the iteration procedure we need to get a "guess" at the potential value for each point in the region. There are many different kinds of initialization schemes that could be used. We could simply set each point equal to zero (on a large, fast computer, e.g. Cray, this is usually done). Or we could average the values on the four boundaries and set each point in the region equal to that value. In the interest of computational speed on the Athena VAX 750, we have chosen an initialization scheme which gives a fairly accurate guess. For a detailed description see Electromagnetics, 3rd edition, by John D. Krauss, pp. 287-293.

Iteration

Having assigned an initial guess we then proceed sequentially through the array, recalculating the potential at each point in the region. We start at the upper left grid point and use the potential at each of the four surrounding points (in the x and y directions) in equation (3b) to recalculate the potential for that point. For example, the potential at the point (1,1) in Figure 2 would be recalculated using the following equation:

$$V_{1,1} = V_b + V_{1,2} + V_{2,1} + V_\ell$$

Then the potential at the point (1,2) would be

$$V_{1,2} = V_\ell + V_{1,3} + V_{2,2} + V_{1,1}$$

In this manner the potentials on the first column of points within the boundary are reassigned. Moving to the next column, we recalculate the potentials similarly, using the four surrounding points, i.e.

$$V_{2,1} = V_{1,1} + V_{2,2} + V_{3,1} + V_b$$

Each point is recalculated in this manner (from bottom to top and left to right). This ends the first iteration. In each successive iteration the value for each point is recalculated.

In iterative methods such as this relaxation method the values obtained converge on the exact solution to the problem. Although these values are only approximate, continued iterations can make the values as accurate as desired.

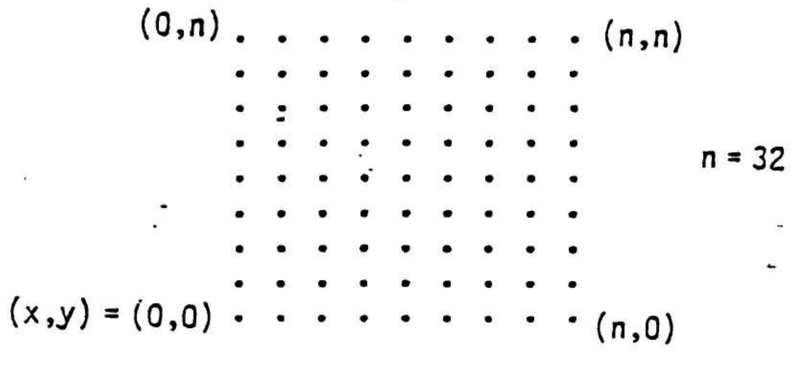


Figure 2

Neumann Boundary Conditions (Normal Derivative of Potential)

The potential within a region can also be determined for Neumann boundary conditions when, instead of the potential (as in Dirichlet boundary conditions), the normal derivative is specified on the boundary. The case we will look at is mixed Neumann-Dirichlet boundary conditions; that is, we are given the potential on three sides and the normal derivative on the remaining side. For this discussion we will place the normal derivative on the left side of the square.

Normal Derivative

All points in the region (boundary and interior points) must satisfy Poisson's equation. The potentials at the left boundary will float until they converge on the unique solution. But in order to converge to their final values, the left boundary points must have four surrounding points so we can use the five-point averaging formula [equation (3)] to solve Poisson's equation. To accommodate this we add an extra column of points just outside the left boundary. During the calculations we will consider this extra column to be the left boundary (see Figure 3).

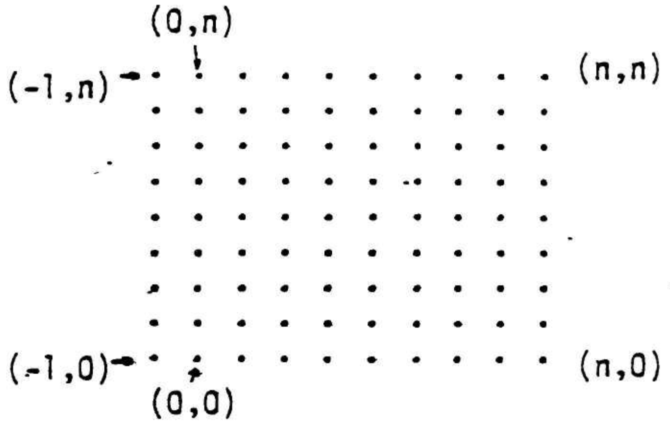


Figure 3

We find potential values for this extra column of points by using the given normal derivatives and the potential values in the first column in the point-slope formula:

$$(V_{-1,k} - V_{1,k}) = m_k(x_{-1} - x_1)$$

where

$V_{-1,k}$	represents the potential at the point on the extra column, kth row;
$V_{1,k}$	represents the potential at the point on the first column, kth row;
m_k	represents the normal derivative on original left boundary and kth row;
x_{-1} and x_1	represent the x position of the extra left and first columns respectively (in Figure 3, $x_{-1} = -1$ and $x_1 = 1$).

We solve for $V_{-1,k}$:

$$V_{-1,k} = V_{1,k} - 2m_k \tag{4}$$

Iteration

Now that all the grid points have initial values we can begin iteration. We recalculate the potential values for each grid point within the region using equation (3) just as we did with the Dirichlet problem. We then readjust the potentials on the extra left column to match the given normal derivative as we did previously using equation (4).

We then sweep through the grid again recalculating all the potentials. These iterations continue until the solution is attained to the desired accuracy. At this point the extra column of potential values on the left is thrown away.

Successive Overrelaxation

In order to arrive at a solution with fewer iterations (i.e., faster) a method called Successive Overrelaxation is used. It is similar to the Relaxation Method except that an additional term is added in equation (3) to account for the error involved in this type of approximation. The interested students can find a description of Successive Overrelaxation (and other methods) in Computer Simulation Using Particles by R. W. Hockney, McGraw-Hill, 1981, p. 179.

Use of Integer Arithmetic

In the interest of gaining even more speed on the Athena VAX 750 machines, calculations are done in integer (rather than floating point) arithmetic. Thus by limiting our accuracy to six decimal figures (which is plenty for a good, qualitative plot) all numbers inside the program are multiplied by 10^6 and declared integer before computations are done. Of course, before displaying the results the final numbers are divided by 10^6.

Electric Field Lines from Conjugate Potentials

If a complex function $\Phi(x, y) = \varphi(x, y) + i\psi(x, y)$ is a solution of Laplace's equation in two dimensions (where φ and ψ are real valued functions of positions x and y), then Φ is an analytic function and satisfies the Cauchy-Riemann relations:

$$\frac{d\varphi}{dx} = \frac{d\psi}{dy} \qquad (5)$$

$$\frac{d\varphi}{dy} = -\frac{d\psi}{dx} \qquad (6)$$

The functions φ and ψ are called conjugate potential functions, and the contours of constant ψ are everywhere orthogonal to the contours of constant φ. If φ is the electrostatic potential, then the contours of constant ψ represent the electric field lines. Note that the converse is also true: ψ may be taken as representing a potential (with appropriate boundary conditions!) and then φ represents the field lines associated with this potential.

Assume now that the potential φ has been calculated on an $n \times n$ grid as described above. Approximating the derivatives in Eq. (5) by the difference formula we get

$$\varphi(i+1, j) - \varphi(i-1, j) = \psi(i, j+1) - \psi(i, j-1) \qquad (7)$$

All of the φ values are known and all of the ψ values are unknown. Since we can calculate the conjugate potential only to within an additive constant, we choose any point [say point (a, b)] and assign $\psi(a, b)$ a value of 0. Then we can find the conjugate potential of all other points in the array relative to $\psi(a, b)$. Hence, using Eq. (7) and substituting in $\psi(a, b)$, we have

$$\varphi(a+1, b-1) - \varphi(a-1, b-1) = \psi(a, b) - \psi(a, b-2)$$

Both φ values are known and $\psi(a, b) = 0$, so we have a linear equation in one unknown and can find $\psi(a, b-2)$. Similarly we can find $\psi(a, b-4)$ and so forth, finding all of the ψ values on row a. Using Eq. (6) in difference form we obtain

$$\varphi(i, j+1) - \varphi(i, j-1) = -[\psi(i+1, j) - \psi(i-1, j)] \tag{8}$$

Substituting $\psi(a, b)$ into Eq. (8),

$$\varphi(a-1, b+1) - \varphi(a-1, b-1) = -[\psi(a, b) - \psi(a-2, b)] \; .$$

Since all potential values are given, we can solve for $\psi(a-2, b)$. Using Eq. (7) repeatedly we can find all of the conjugate potentials in row $a-2$. Similarly the conjugate potential at the points in every row and column can be determined. This array is plotted as the field lines on top of the contour plot of potential.

Attachment 2 to POISSON Program

WHAT IS A REASONABLE SIZE FOR $\left|\frac{dV}{dn}\right|$?

The Poisson program allows one to specify Neumann conditions on any or all of the boundaries, that is, to specify the normal derivative of the potential dV/dn. When prompted to supply a numerical value for this normal derivative, we are tempted to use (for lack of a better number) the all-purpose "unity", only to find that the program slows to a halt for $|dV/dn| \sim 0(1)$. It is easy to understand this slow down if we realize that the program normalizes all lengths to the distance between neighboring grid points on the 33×33 grid. Hence, $|dV/dn| = 1$ is a very large gradient ($\Delta V \sim 1$ between neighboring grid points).

To find an appropriate value for $|dV/dn|$, such that all gradients will be gradual and the convergence will be quick, we require that the normal gradient scale length

$$\frac{1}{L_n} \equiv \left|\frac{1}{V}\frac{dV}{dn}\right|$$

be of the same order of magnitude as the characteristic length of our bounded region ($\ell = 32$). For potentials of order unity, this gives

$$\left|\frac{dV}{dn}\right| \sim \frac{1}{32} \sim 0.1 \text{ to } 0.01\ .$$

Specified normal derivatives of this order are consistent with our choice of default error bound criteria, and will result in very quick calculation of the potentials.

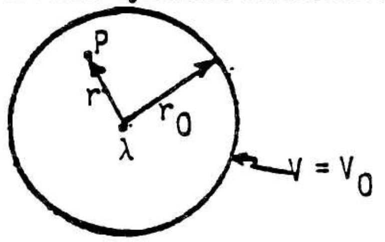
Attachment 3 to POISSON Program

WHAT IS A "GOOD CHOICE" FOR λ?

If we "carelessly" toss an overly-large line-charge into our square region, then the potential inside the region will be dominated by the influence of that line-charge, and the specific values chosen for the boundary conditions on the region will have very little impact on the final solution. This slows down the relaxation process. The reasoning is as follows:

In a 32×32 grid, the influence of the boundary conditions is leaked into the 30×30 interior points of the grid from the 120 (non-corner) boundary points. This is a fairly quick process. On the other hand, the influence of the line-charge on the potentials is leaked out slowly from a single source point. For example, after 4 relaxation iterations (assuming the simplest relaxation scheme) the influence of the boundaries has made an impact at 416 internal grid points, while the influence of the line-charge has been felt by at most only 29 points.

What, then, is a reasonable range of values for the line-charge density λ? Consider a line-charge along the axis of a circular cylinder of radius r_0.



If we assume that the potential on the boundary is a constant, say $V = V_0$, then the potential inside the cylinder at a field point P is:

$$V = V_0 - \frac{\lambda}{2\pi\epsilon_0} \ln\left(\frac{r}{r_0}\right)$$

where ϵ_0 is the permitivity of free space, and r is the distance between P and the line-charge. In order for the specified boundary conditions to have a significant influence, and yet still be able to see the effects of the line-charge, we want the difference between the potential close to the line-charge and the potential on the boundary to lie approximately in the range:

$$1 \text{ Volt} \lesssim V - V_0 \lesssim 10 \text{ Volts} \qquad \text{for } V \text{ "near" the line charge.}$$

How near is "near"? If one gets too near ($r \to 0$) then $V \to \infty$. The closest we can get, however, is one grid point away, or $r = 1$. For our square 32×32 problem, a reasonable estimate of r_0 is $r_0 = 16$. We may now give an order-of-magnitude estimate of λ, such that the line-charge will not dominate the calculation of the potential:

$$1 \text{ Volt} \lesssim \frac{-\lambda}{2\pi\epsilon_0} \ln\left(\frac{1}{16}\right) \lesssim 10 \text{ Volts}$$

or

$$2.0 \times 10^{-11} \lesssim |\lambda| \lesssim 2.0 \times 10^{-10} .$$

Note: The units of λ are Coulombs per unit length in the direction perpendicular to the plane of the square region.

Note also: The above discussion is related to a suggested range of potentials. Should you wish to scale up the potentials (by say two-orders of magnitude) the λ's should also be scaled up similarly. In addition the abserr (see "Explanation of Error Criteria for the Poisson Program") should be scaled up likewise.

Attachment 4 to POISSON Program

EXPLANATION OF ERROR CRITERIA FOR THE POISSON PROGRAM

The algorithm used to compute the potential values is called SOR, or Successive Overrelaxation. It is an iterative algorithm so one must specify how accurate a solution is desired to consider the problem solved. However, requiring a relatively stringent set of error bounds may lead to an unreasonable amount of computation time. If a set of error bounds is not chosen stringently enough, then the accuracy of the solution may be degraded to an unacceptable level. The smoothness and regularity of the contours that the contour-plotter produces is a good indicator of acceptable or unacceptable accuracy. Because of this tradeoff between computation time and accuracy, Poisson has error bounds as input variables.

The structure of the error criteria is as follows: if *either* an absolute tolerance or a relative tolerance is met, then the problem is considered solved. Thus, there are two error bound variables, the relative error (relerr) and the absolute error (abserr).

At a given point within the region, the difference between the current value at that point and its previous value is divided by the previous value. This computes the percentage change of the current value. The absolute value of this quantity is compared to the value of the relative error bound. The absolute value of the difference between the current value and the previous one is also compared to the value of the absolute error bound.

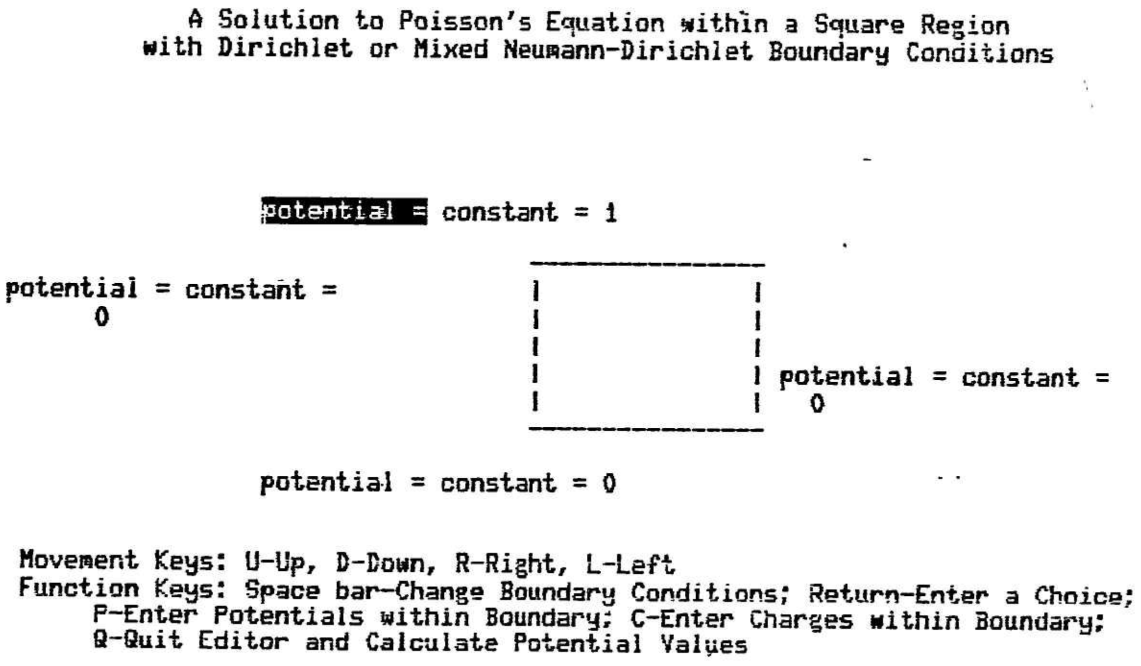
The default setting is relerr = 1e-02 and abserr = 1e-02. Note that the absolute error bound default is reasonable for potentials in the range ± 1 Volt to ± 10 Volts.

Note: If potentials and λ's are scaled up from the above suggested ranges, the abserr should also be similarly scaled.

Appendix B

Problems with the old user interface.

Figure B-1 is what the main screen of the old "Poisson" interface looked like.



```
               A Solution to Poisson's Equation within a Square Region
             with Dirichlet or Mixed Neumann-Dirichlet Boundary Conditions


                      potential = constant = 1
                                            ----------------
potential = constant =                      |              |
     0                                      |              |
                                            |              |
                                            |              | potential = constant =
                                            |              |   0
                                            ----------------

                      potential = constant = 0

Movement Keys: U-Up, D-Down, R-Right, L-Left
Function Keys: Space bar-Change Boundary Conditions; Return-Enter a Choice;
     P-Enter Potentials within Boundary; C-Enter Charges within Boundary;
     Q-Quit Editor and Calculate Potential Values
```

Figure B-1: This is the main screen of the old interface.

Shown are the default values for each of the four boundaries. The default problem box has Dirichlet boundary conditions. All of the boundaries except the top one are tied to zero volts; the top boundary is tied to 1.0 volts.

There are several minor problems and one major problem with this old interface. Before getting to the major problem, I'll discuss the minor ones.

B.1 Lack of a help command.

First, there is no "help" command. The only commands available are: (1)enter potentials, (2)enter line charges, (3)change boundary conditions, and (4)calculate the answer. For the first-time user, it's comforting to know that help is available on-line. A help command which just prints a paragraph (giving an overview of the purpose of the program, and how it works) is a lot better than no help command at all.

B.2 The boundary conditions are confusing.

The second problem with the main screen is that the boundary conditions are confusing. Why are the numeric values (see the zeros and the one in Figure B-1) positioned so awkwardly? Do these values have to be integers? [Answer: No!] If not, then why are they shown as integers?

B.3 The problem box is confusing.

The third problem with the old interface is with the graphical representation of the problem box. Why are there gaps in the walls of the box? Are they meaningful, i.e., are they supposed to symbolize something? [Answer:

No!] Why aren't they walls of the box drawn with solid lines? The user is apt to think that the gaps DO mean something, because as soon as he solves his first problem (and sees the equipotential and E fields being plotted with smooth, continous lines), he'll realize that high resolution graphics ARE available on this computer terminal, which implies that the problem box COULD have been drawn with smooth lines.

B.4 Moving the high-lighted region is awkward.

The fourth problem deals with moving the high-lighted region about the screen. The high-lighted region is sort of like a pointer, in that it physically points to where the next command will take place (this will become more meaningful when I discuss changing the boundary conditions). The problem is that in order to move the high-lighted region,the user must type "u", "d", "l", or "r" to move up, down, left, and right, respectively. For the user who isn't a touch typist, this can be a major bottleneck because it forces him to hunt around the keyboard for these keys; they're scattered all over the keyboard! A better scheme would have been to pick four keys which are right beside each other, and are clustered in some logical fashion [see Figure B-2].

The scheme proposed in Figure B-2 has the advantage that once the user locates the cluster of keys on the

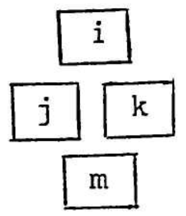


Figure B-2: An alternative to the u, d, l, and r keys.

keyboard, he doesn't need to look at the keyboard any more to find the key he wants. However, this scheme has the drawback that using "i", "j", "k", and "m" when you really mean "up", "left", "right",and "down" is not intuitive. The first-time user wouldn't look at the keyboard and think "Oh, you probably use the i, j, k, and m keys to move the high-lighted region." Rather, he would think "Oh, you probably use the arrow keys to move the high-lighted region.". Of course, a mouse would be even more intuitive, but the terminal that this program runs on (a DEC VT240) doesn't have a mouse.

B.5 It's hard to figure out how to change the boundary conditions.

The fifth problem is figuring out how to change the boundary conditions on the problem box. The text at the bottom of the screen [see Figure B-1] gives the user the following cryptic advice: "Space bar-Change Boundary Conditions; Return-Enter a Choice".

B.5.1 Explanation of how to change the boundary conditions, with the old interface.

Remember that for each of the four boundaries, there are three things that the user must do. He must:

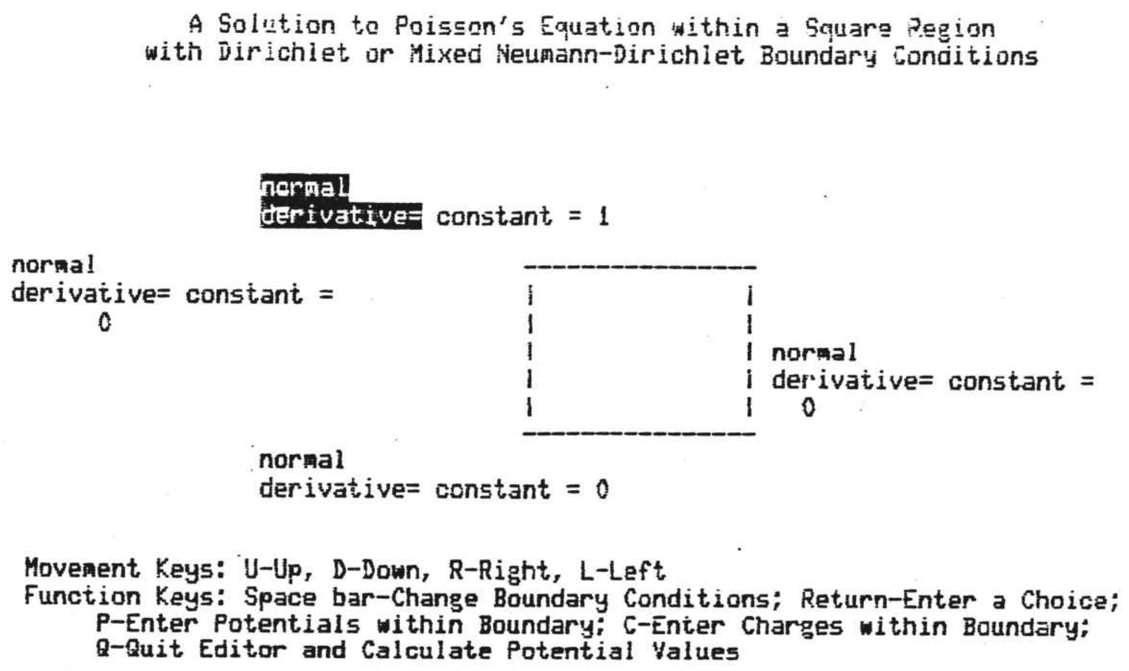
(1)choose whether to specify the potential at the boundary xor to specify the normal derivative of the potential at that boundary.

(2)choose one of five functions (constant, linear, step, sine, or cosine) for that boundary.

(3)specify the numerical parameters for that function.

The default problem [see Figure B-1] has the potential of each boundary specified. If the user wants to specify the normal derivative of the potential instead, then he would: (1)move the high-lighted region until the word "potential" (by the desired boundary) is high-lighted, and then (2)hit the space bar. This will cause the word "potential" to disappear, and be replaced with the high-lighted words "normal derivative". If the space bar is hit AGAIN, then the potential/normal derivative option will be toggled back to "potential", and the word "potential" will again appear (in the place of "normal derivative") [see Figure B-3].

For each boundary, the user must choose one of the five functions. To do this, he moves the high-lighted region to one of the boundaries, where the word "constant"



```
              A Solution to Poisson's Equation within a Square Region
          with Dirichlet or Mixed Neumann-Dirichlet Boundary Conditions


                      normal
                      derivative= constant = 1

normal                                      ----------------
derivative= constant =                      |              |
     0                                      |              |
                                            |              | normal
                                            |              | derivative= constant =
                                            |              |   0
                                            ----------------
                      normal
                      derivative= constant = 0

 Movement Keys: U-Up, D-Down, R-Right, L-Left
 Function Keys: Space bar-Change Boundary Conditions; Return-Enter a Choice;
      P-Enter Potentials within Boundary; C-Enter Charges within Boundary;
      Q-Quit Editor and Calculate Potential Values
```

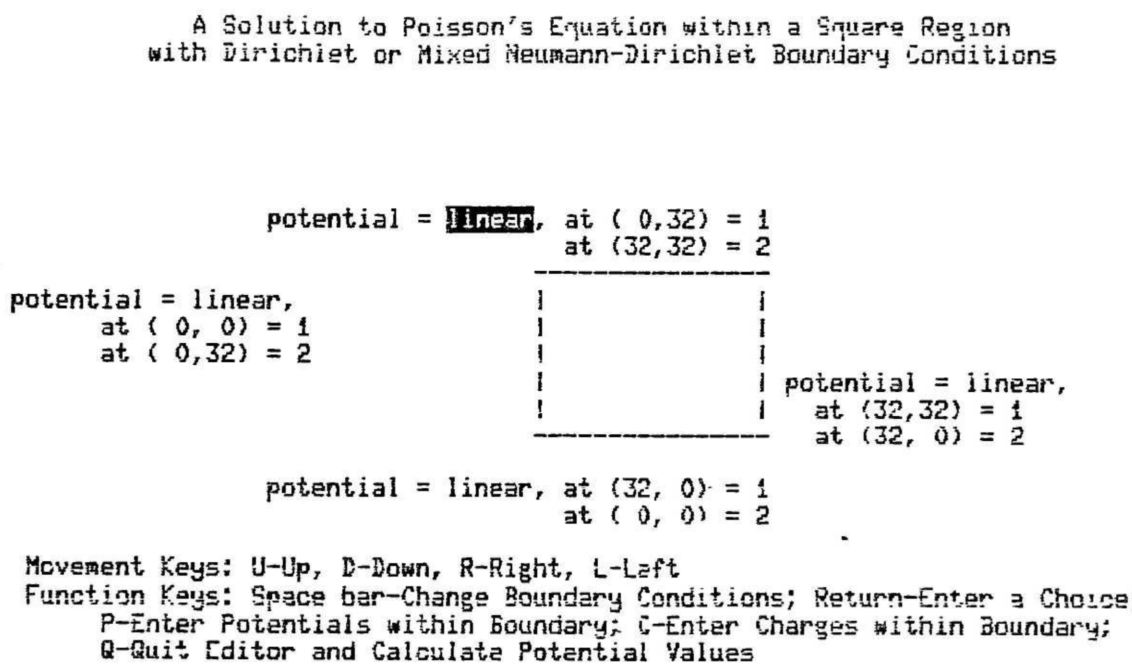
Figure B-3: Top boundary toggled to "normal derivative".

is written. If he hits the space bar, then the line "constant = 0" will be replaced with:

```
linear, at ( 0,32) = 1
        at (32,32) = 2
```

This means that the potential (xor it's normal derivative) will start at a value of 1 (at the upper left corner of the problem box, at (x,y) = (0,32)), and increase linearly to a value of 2 (at the upper right corner of the problem box, at (x, y) = (32,32)) [see Figure B-4].

Hitting the space bar four more times will toggle to the step option, the sine option, the cosine option, and then back to the constant option.

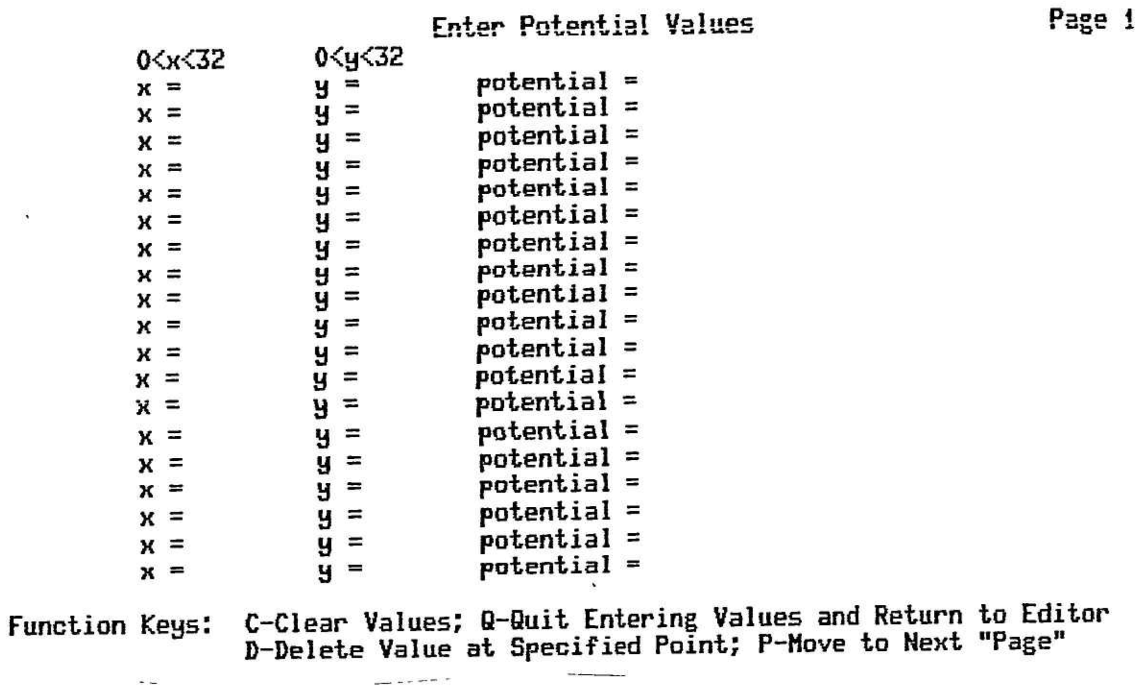


```
            A Solution to Poisson's Equation within a Square Region
          with Dirichlet or Mixed Neumann-Dirichlet Boundary Conditions



                  potential = linear, at ( 0,32) = 1
                                      at (32,32) = 2
                                    ----------------
potential = linear,                 |              |
      at ( 0, 0) = 1                |              |
      at ( 0,32) = 2                |              |
                                    |              | potential = linear,
                                    |              |   at (32,32) = 1
                                    ----------------   at (32, 0) = 2

                  potential = linear, at (32, 0) = 1
                                      at ( 0, 0) = 2

 Movement Keys: U-Up, D-Down, R-Right, L-Left
 Function Keys: Space bar-Change Boundary Conditions; Return-Enter a Choice;
      P-Enter Potentials within Boundary; C-Enter Charges within Boundary;
      Q-Quit Editor and Calculate Potential Values
```

Figure B-4: All boundaries are "linear".

B.6 The MAIN PROBLEM with the old interface.

The main problem with the old interface is the barbaric way that data was entered, i.e., the information which conveyed the position and the value of line charges, sheets of constant potential, and rods of constant potential. The user had to fill up a table [see Figure B-5]. This was an incredible pain it the neck because it was hard to use, and because it forced the user to enter much more data than necessary. In particular, a line is completely determined by its two endpoints -- the user shouldn't have to specify all of it's interior points as well! Also, for a problem of moderate complexity, the



```
                         Enter Potential Values                    Page 1
       0<x<32     0<y<32
       x =        y =        potential =
       x =        y =        potential =
       x =        y =        potential =
       x =        y =        potential =
       x =        y =        potential =
       x =        y =        potential =
       x =        y =        potential =
       x =        y =        potential =
       x =        y =        potential =
       x =        y =        potential =
       x =        y =        potential =
       x =        y =        potential =
       x =        y =        potential =
       x =        y =        potential =
       x =        y =        potential =
       x =        y =        potential =
       x =        y =        potential =
       x =        y =        potential =
       x =        y =        potential =

Function Keys:  C-Clear Values; Q-Quit Entering Values and Return to Editor
                D-Delete Value at Specified Point; P-Move to Next "Page"
```

Figure B-5: Hard way of entering potentials.

user is forced to first plot the problem on graph paper. He must then manually enter all of this data into the table in Figure B-5.

Appendix C

The hardware that the "Poisson" program runs on.

The "Poisson" program requires the following hardware in order to run properly:

1)a DEC VAX-11/750, running a UNIX operating system.

2)a DEC VT125 color monitor (model VR41-A). A DEC VT125 monochrome monitor (model VR201) will also work.

3)a DEC VT240 graphics unit (model VS-240A). This is a rectangular box which sits beneath the monitor. It is needed to interpret the ReGIS graphics commands (that are sent to the terminal by my program).

Appendix D

Explanation of coordinate mapping.

It is often necessary to map a point from the interior of a rectangle to a point in the interior of another rectangle. One place where this is required is when going from the graphics editor screen to the main screen (see Figure D-1). Here, two sheets of constant potential are entered on the graphics editor screen, and then are redisplayed when the user returns to the main screen.

D.1 The problem.

The $64,000.00 question is: how does one map a line (drawn in the problem box) on the graphics editor screen to a line (drawn in the problem box) on the main screen? This is not a trivial question; the mapping must be done is such a way that the relative proportion and position of the line is the same in both problem boxes.

D.2 The solution.

A line, l_1, is defined by its two endpoints, $P_1(x,y)$ and $P_2(x,y)$. Thus, if we could figure out how to map a point, $P(x,y)$, from one rectangle, R, to an equivalent point $P'(x,y)$ in another rectangle, R', then we could easily map a line, or even a polygon (which is just a set of lines), from one rectangle into another rectangle.

Consider Figure D-2. We want to preserve relative proportions. This means that the distance of the point from the y boundary to the length of the x boundary, and the ratio of the distance of the point from the x boundary to the length of the y boundary, must be the same in screen A as screen B (see Figure D-2).

In terms of Figure D-2, this results in the following constraints:

$$\frac{x_{A,dist}}{x_{A,length}} = \frac{x_A - x_{A,min}}{x_{A,max} - x_{A,min}} = \frac{x_B - x_{B,min}}{x_{B,max} - x_{B,min}} = \frac{x_{B,dist}}{x_{B,length}}$$

and

$$\frac{y_{A,dist}}{y_{A,length}} = \frac{y_A - y_{A,min}}{y_{A,max} - y_{A,min}} = \frac{y_B - y_{B,min}}{y_{B,max} - y_{B,min}} = \frac{y_{B,dist}}{y_{B,length}}$$

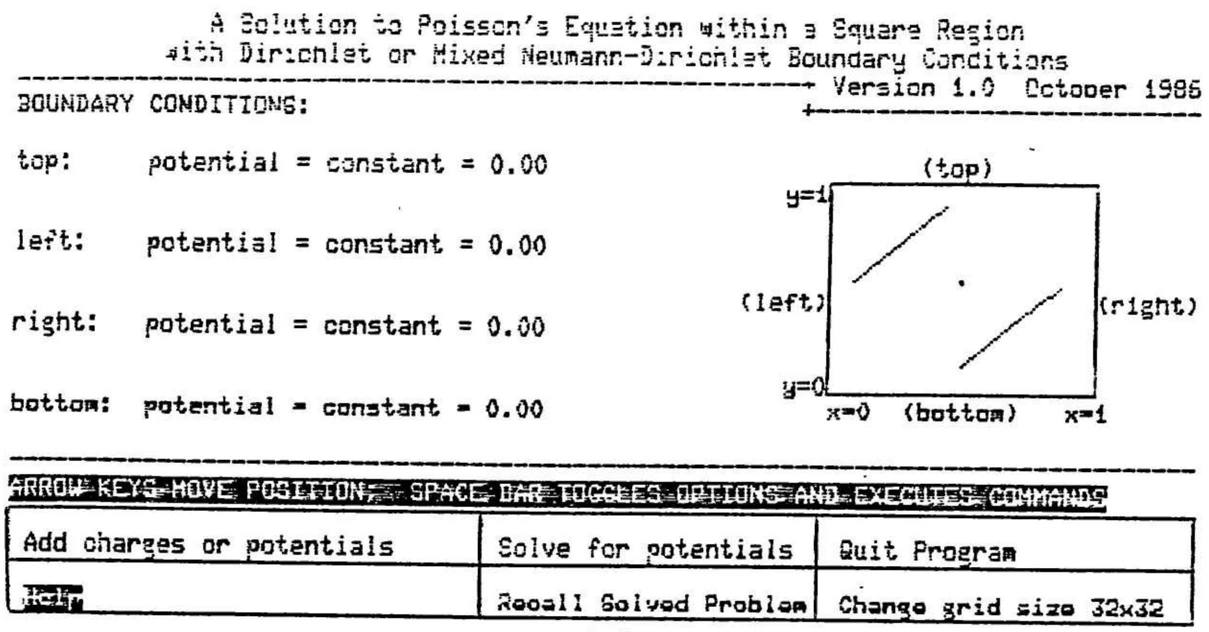
These equations can be reduced to the following mapping equations:

$$x_B = x_{B,min} + \frac{x_{B,max} - x_{B,min}}{x_{A,max} - x_{A,min}}(x_A - x_{A,min})$$

$$y_B = y_{B,min} + \frac{y_{B,max} - y_{B,min}}{y_{B,max} - y_{B,min}}(y_B - y_{B,min})$$

This results in the C subroutine shown in Figure D-3, which takes the coordinates of two rectangles, and maps a point in the interior of one of them to a point in the interior of the other rectangle.

There are three screens that the ES problem has to be mapped to: (1)the graphics editor screen, (2)the main screen, and (3)the contour plotting screen. See Figure D-4.



A Solution to Poisson's Equation within a Square Region
with Dirichlet or Mixed Neumann-Dirichlet Boundary Conditions

Version 1.0 October 1986

BOUNDARY CONDITIONS:

top: potential = constant = 0.00

left: potential = constant = 0.00

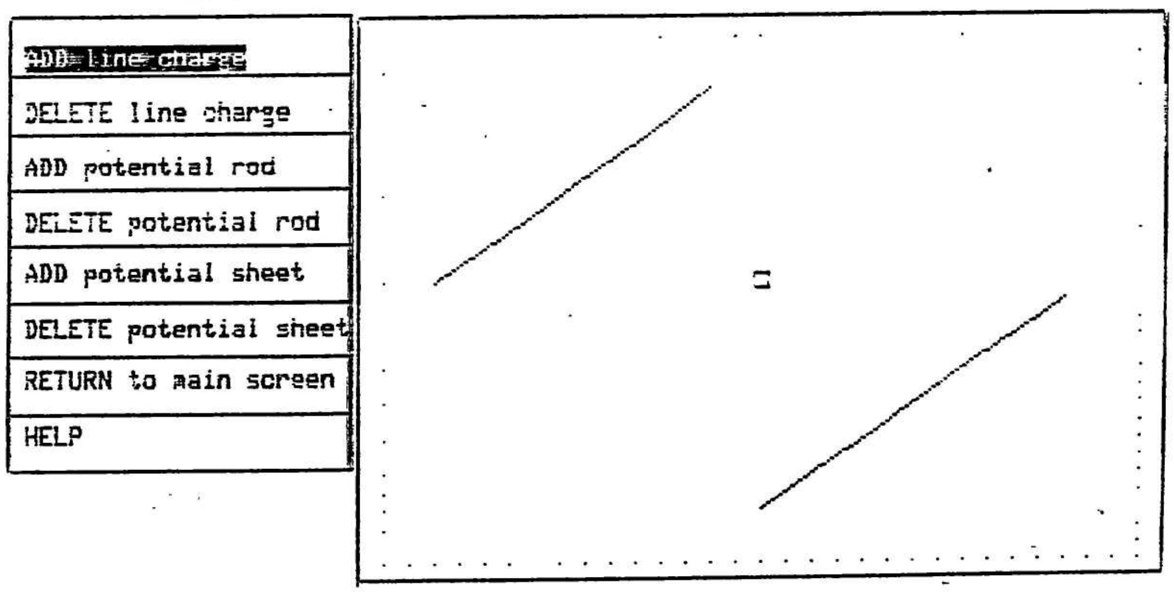
right: potential = constant = 0.00

bottom: potential = constant = 0.00

ARROW KEYS MOVE POSITION; SPACE BAR TOGGLES OPTIONS AND EXECUTES COMMANDS

Add charges or potentials	Solve for potentials	Quit Program
Help	Recall Solved Problem	Change grid size 32x32

(a)



(b)

Figure D-1: Two sheets of constant potential and one line charge, shown in (a)the main screen, and in (b)the graphics editor screen.

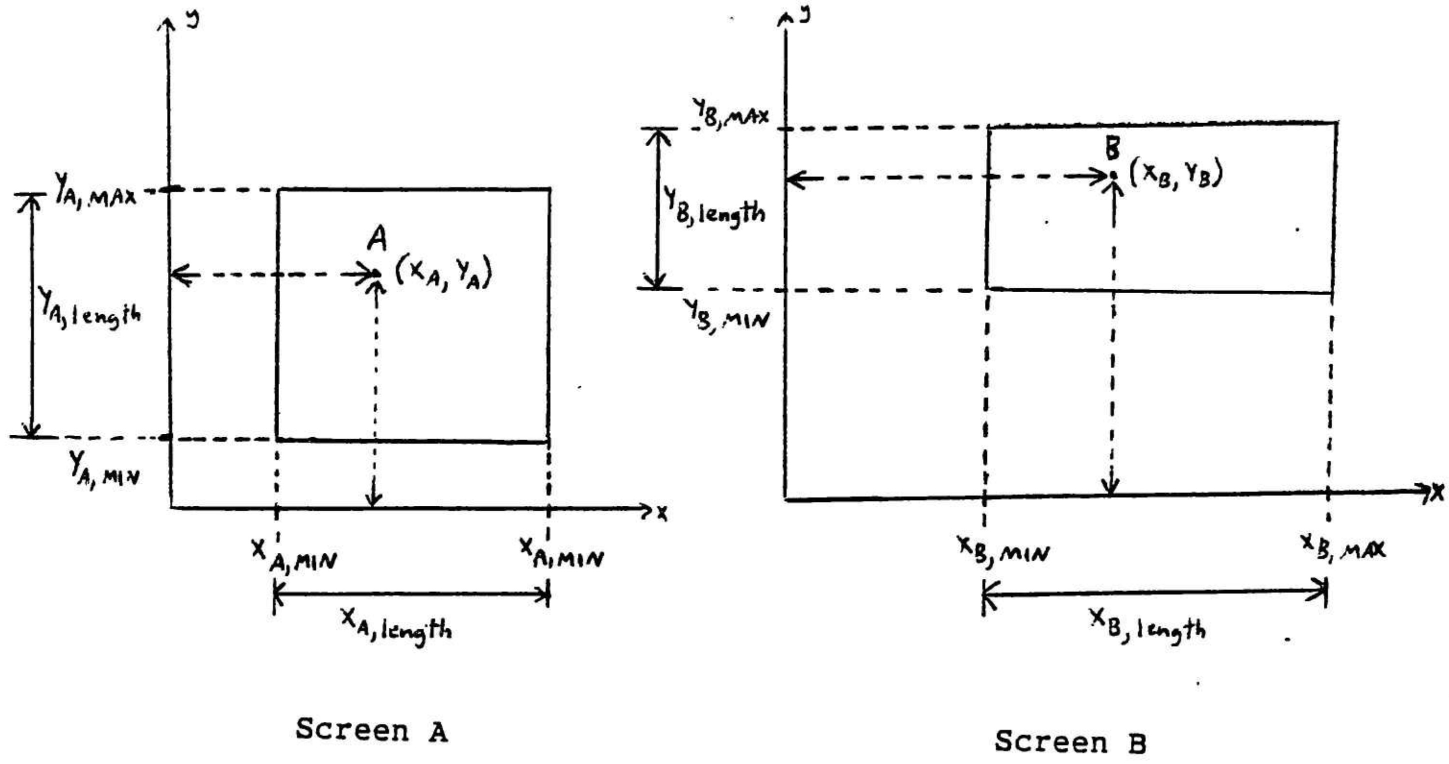
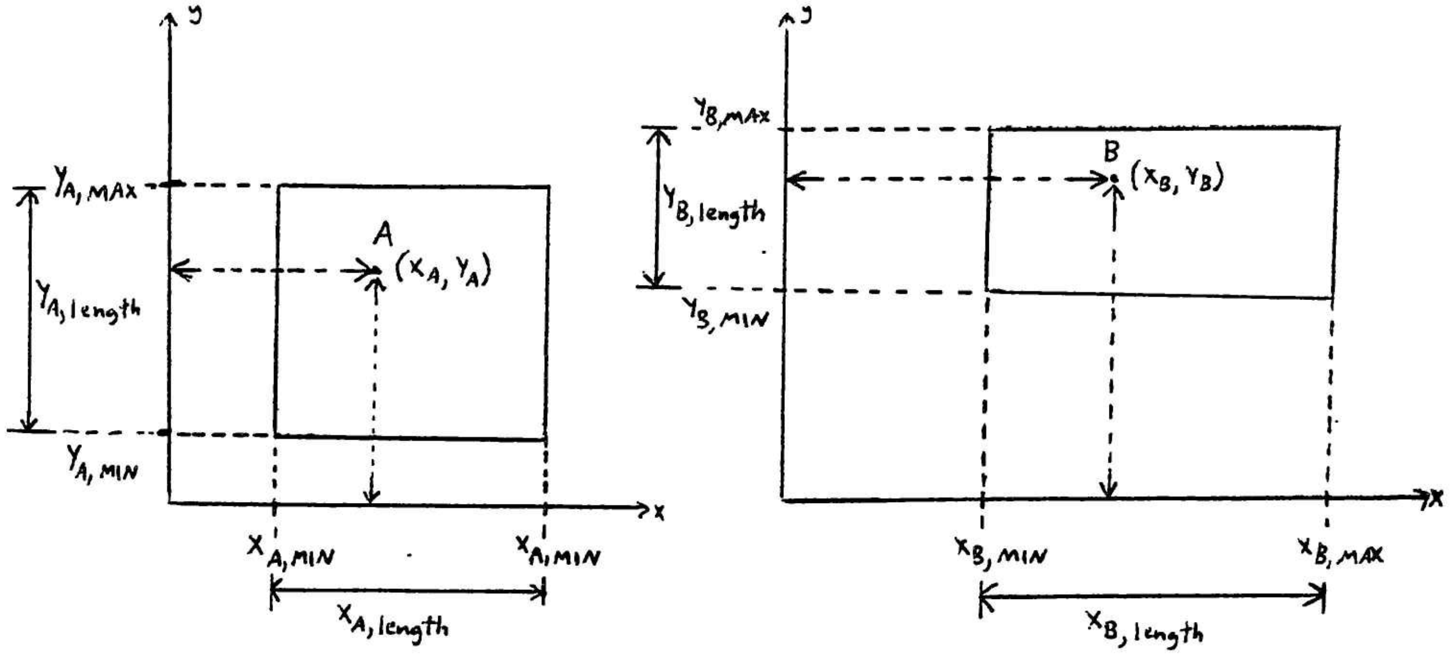


Figure D-2: How do you find an expression for the coordinates of point B, i.e., (X_B, Y_B), when you are given: the coordinates of point A, (X_A, Y_A), and the max and min coordinates of the two rectangles?



```
map_point(Xa_max, Xa_min, Ya_max, Ya_min,
          Xb_max, Xb_min, Yb_max, Yb_min,
          Xa, Ya, Xb, Yb)
double Xa_max, Xa_min, Ya_max, Ya_min;
double Xb_max, Xb_min, Yb_max, Yb_min;
double Xa, Ya, Xb, Yb;
{
        *Xb = Xb_min + (Xb_max - Xb_min)/(Xa_max - Xa_min)
                      *(Xa - Xa_max);

        *Yb = Yb_min + (Yb_max - Yb_min)/(Ya_max - Ya_min)
                      *(Ya - Ya_max);
}
```

Figure D-3: A general point-mapping subroutine (written in C). Given the point (X_a, Y_a), and the max and min coordinates of each problem box, this subroutine returns the correct values for (X_b, Y_b).

Appendix E

The contour plotter used in "Poisson".

Roberta Brawer
July 18, 1985

MEMO ON CALLING THE CONTOUR PLOTTING ROUTINE

1. Introduction

The contour plotting package accepts a two-dimensional array of double precision numbers and draws a specified number of constant value contours spaced by equal intervals. The plot will be centered in the graphics region of the terminal display.

There are two routines that may be called - setup and contour_plot. Setup initializes the Penplot graphics package, sets the display the selected coordinate scales, and draws an outline box around the plotting region. The routine contour_plot is used to actually find and draw the contours.

The contour routines are written in 'C' and are designed to be called from 'C'. (For details about calling the routines from Fortran, please refer to Section 6.4.) The actual plotting on the terminal display, though, is done by using the Penplot graphics library which is written in Fortran.

Using Penplot, additional features, such as boundaries, can be drawn directly on the same graph as the contours. Direct calls to Penplot routines can be freely interspersed with calls to the contour plotter (provided one follows the correct proctocol for calling Fortran subroutines from 'C'). For information on Penplot, consult the "Penplot Graphics Control System Reference Manual," available in the Athena clusters.

To insure that no scaling distortion occurs, the contour program automatically sets equivalent scales on the horizontal and vertical axes. By equivalent scales, it is meant that if one unit of length corresponds, on the x axis, to, say, one centimeter on the screen, then it will also correspond, on the y axis, to one centimeter on the screen. (If the scales were not equivalent, then, for example, circles would be drawn as ellipses.)

As a consequence of this coupled scaling, the length of the horizontal or vertical side of the plot will be proportional to the difference between the values of the maximum and minimum coordinates along the x or y axis, respectively. If this difference is unequal for the two axes, the plot will occupy a rectangular region.

2. Calling setup:

The setup routine provides graphics initialization for Penplot, sets up the coordinate system, and draws a box outlining the plotting area. It must be called before contour_plot unless the initialization has been handled by direct calls to the Penplot library.

After the outline is drawn in the selected color, the pen will automatically be set to green for the contours. If a different color for the contours is preferred, use a Penplot pen command between calls to setup and contour_plot.

If setup is not used, and initialization is done directly with Penplot, it is preferable to use the routine, show(), rather than area(), in establishing the coordinate scales. This insures that the axes are scaled equivalently, not independently, thus avoiding geometric distortion.

Routine: setup(left, right, bottom, top, color)

A guments:

left	[double]	x coordinate of left side of plotting area
right	[double]	x coordinate of right side of plotting area
bottom	[double]	y coordinate of bottom of plotting area
top	[double]	y coordinate of top of plotting area
color	[long]	pen color of the outline box: blue = 1, red = 2, green = 3

---

3. Calling contour plot:

The routine contour_plot accepts a two dimensional array of numbers and plots a selected number of equivalue contours. The array must have a double precision data type and must be contigous in memory since only the address of the first element is passed. The maximum resolution for the array is [101] x [101]. Both the number of horizontal and vertical grid points, NX and NY, must be less than or equal to 101.

The ordering of the indices in the array is in spatial analogy with the matrix notation convention of listing the row index first and the column index second. Thus, the first index identifies the vertical location and the second index, the horizontal location. Please note that this is opposite to the Cartesian coordinate convention. In other words, the value at the coordinate point (xindex, yindex) would be represented by the array element, array[yindex][xindex].

Routine: contour_plot(array, NX, NY, left, right, top, bottom, ncontours, minvalue, maxvalue)

Arguments:

array	[array name]	address of first element of the array
NX	[integer]	number of horizontal grid points
NY	[integer]	number of vertical grid points
left	[double]	x coordinate of left side of plotting area
right	[double]	x coordinate of right side of plotting area
bottom	[double]	y coordinate of bottom of plotting area
top	[double]	y coordinate of top of plotting area
ncontours	[integer]	number of contours to be plotted
minvalue	[double]	minimum contour value to be plotted
maxvalue	[double]	maximum contour value to be plotted

---

4. Example

```
#define NX 51
#define NY 51
#define BLUE 1
#define RED 2
#define GREEN 3
#define BIG 1.0e20

main()           /* example program for contour plotter */
{
    double array[NY][NX], result;
    double xmin, xmax, ymin, ymax, offset;
    long color;
    int ncontours;
    double x, y, minvalue, maxvalue;
    int xindex, yindex;

    xmin = -25.0;        /* establish coordinate system */
    xmax = 25.0;
    ymin = -25.0;
    ymax = 25.0;
    offset = 25.0;

    minvalue = BIG;      /* initialize to large number */
    maxvalue = -BIG;     /* initialize to small number */

    /* Load array. Note: indices must be positive integers. */
    for(yindex = 0; yindex < NY; yindex++)
        {
        y = yindex - offset;    /* origin at center of the plot */
        for(xindex = 0; xindex < NX; xindex++)
            {
            x = xindex - offset;
            result = x*x + 2.0*y*y;

            if (result > maxvalue)      /* find maximum value */
                maxvalue = result;
            if (result < minvalue)      /* find minimum value */
                minvalue = result;

            array[yindex][xindex] = result;
            }
        }

    /* Call contour routines */
    color = RED;
    ncontours = 11;

    setup(xmin, xmax, ymin, ymax, color);
    contour_plot(array, NX, NY, xmin, xmax, ymin, ymax, ncontours,
                 minvalue, maxvalue);

}
```

5. Linking Instructions

The object code for the contour routines is located in an archive, /projects/6_d0004/lib/contour.a, on aphrodite and hactar. In addition, the Penplot, Fortran, and math libraries are all needed. Therefore, include the following libraries in the linking command: /projects/6_d0004/lib/contour.a -lpenplot -lF77 -lI77 -lU77 -lX -lm

For example:

```
cc -o program program.c /projects/6_d0004/contour.a -lpenplot
-l{F,I,U}77 -lX -lm
```

6. Additional Notes

6.1. Blank Regions in Plot

If the absolute value of a number in the array is greater than 1.0e12, no contour will be drawn through that point or through a grid segment with that point as one of its endpoints. If desired, this feature can be used to blank out regions of the plotting area.

6.2. Handling Singularities

Singular points, or infinities, can be represented in the array as a very large number. It is recommended that the absolute value of the number representing such a point should be less than 1.0e12, but greater than 1.0e10. This will insure that the contours near that point will be closed (see above) and that the electric field plotting program will know that the field lines end at such points. (The electric field plotting program uses values greater than 1.0e10 to identify charge singularities.)

6.3. Discontinuous Functions

The contour program can be used to plot functions which are discontinuous across interfaces. For example, electric field lines across dielectric boundaries can be plotted by finding, in each region, the conjugate harmonic function to the potential whose contours represent the electric field lines.

The plotting can be done by breaking the problem into separate regions, blanking out all but the region to be plotted, and generating multiple calls to the contour plotter. The value of -1 in the argument ncontours indicates that this is a repeat call and that the interval between adjacent contours should be the same as in the previous call. (For an example, see the source code, dielectric-cylinder.c).

6.4. Calling the Contour Routine from Fortran

Since the arguments in the routines are passed by value, not address, the contour plotter cannot be called directly from Fortran. (Fortran automatically passes arguments by address.) To get around this problem, I suggest the following.

Write a short middleman program in 'C' which simply translates the way the arguments are passed. Have the Fortran program call the middleman program, which is designed to receive its arguments as addresses, or pointers. Then the middleman program calls the contour routines, now passing the arguments by value. (Please note that this only concerns the arguments other than the array itself. In both Fortran and 'C', arrays are always passed as the address of the first element of the array.)

6.5. Bugs to be Corrected

The program does not currently deal properly with the situation in which contour lines of the same value cross each other. In such a case, an artefact of a diamond or branch is produced. Hopefully, this will be fixed shortly.

Appendix F

Notes for future programmers.

F.1 How to call C Subroutines from FORTRAN77.

As is noted in the source code documentation, if you want to call a subroutine written in C from a FORTRAN77 program, then the C subroutine must have an underscore appended to it's name.

Example: If the C subroutine call called "foo", then in the C source file it would be defined as:

```
foo_()
{
        printf("\nFoo on you too.");
}
```

And it would be called from FORTRAN as:

```
call foo
```

F.2 How to compile a program that is made up of C source code and FORTRAN77 source code.

All of the files must either have FORTRAN77 source code or C source code in them; you can't mix the two languages within the same file. You start off by compiling all of the separate source code files. Halt the compilation at the object code level; DON'T compile all the way to executable code!

Example: To compile a C file called "foo.c" you would type:

```
cc -o foo.c
```

This would produce the object code file "foo.o".

To compile a FORTRAN77 file called "goo.f" you would type:

```
f77 -c goo.f
```

This would produce the object code file "goo.o".

The next step is to link together all of the object code files, along with the proper libraries. It doesn't matter what compiler you do this with, both "cc" and "f77" will work.

NOTE: If the file "boo.c" is "include"d in the file "moo.c", then you DON'T have to separately compile "boo.c". Just compile "moo.c".

Appendix G

The ASCII character set.

ASCII Character Set

Graphic	Decimal Value	Comments
	0	Null
	1	Start of heading
	2	Start of text
	3	End of text
	4	End of transmission
	5	Enquiry
	6	Acknowledge
	7	Bell
	8	Backspace
	9	Horizontal tabulation
	10	Line feed
	11	Vertical tabulation
	12	Form feed
	13	Carriage return
	14	Shift out
	15	Shift in
	16	Data link escape
	17	Device control 1
	18	Device control 2
	19	Device control 3
	20	Device control 4
	21	Negative acknowledge
	22	Synchronous idle
	23	End of transmission block
	24	Cancel
	25	End of medium
	26	Substitute
	27	Escape
	28	File separator
	29	Group separator
	30	Record separator
	31	Unit separator
	32	Space
!	33	Exclamation point
"	34	Quotation mark

Graphic	Decimal Value	Comments
#	35	Number sign
$	36	Dollar sign
%	37	Percent sign
&	38	Ampersand
'	39	Apostrophe
(	40	Opening parenthesis
)	41	Closing parenthesis
*	42	Asterisk
+	43	Plus
,	44	Comma
-	45	Hyphen (Minus)
.	46	Period (Decimal)
/	47	Slant
0	48	Zero
1	49	One
2	50	Two
3	51	Three
4	52	Four
5	53	Five
6	54	Six
7	55	Seven
8	56	Eight
9	57	Nine
:	58	Colon
;	59	Semicolon
<	60	Less than
=	61	Equals
>	62	Greater than
?	63	Question mark
@	64	Commercial at
A	65	Uppercase A
B	66	Uppercase B
C	67	Uppercase C
D	68	Uppercase D
E	69	Uppercase E
F	70	Uppercase F
G	71	Uppercase G
H	72	Uppercase H
I	73	Uppercase I
J	74	Uppercase J
K	75	Uppercase K
L	76	Uppercase L
M	77	Uppercase M
N	78	Uppercase N
O	79	Uppercase O
P	80	Uppercase P
Q	81	Uppercase Q
R	82	Uppercase R

Graphic	Decimal Value	Comments
S	83	Uppercase S
T	84	Uppercase T
U	85	Uppercase U
V	86	Uppercase V
W	87	Uppercase W
X	88	Uppercase X
Y	89	Uppercase Y
Z	90	Uppercase Z
[	91	Opening bracket
\	92	Reverse slant
]	93	Closing bracket
^	94	Circumflex
_	95	Underscore
`	96	Grave accent
a	97	Lowercase a
b	98	Lowercase b
c	99	Lowercase c
d	100	Lowercase d
e	101	Lowercase e
f	102	Lowercase f
g	103	Lowercase g
h	104	Lowercase h
i	105	Lowercase i
j	106	Lowercase j
k	107	Lowercase k
l	108	Lowercase l
m	109	Lowercase m
n	110	Lowercase n
o	111	Lowercase o
p	112	Lowercase p
q	113	Lowercase q
r	114	Lowercase r
s	115	Lowercase s
t	116	Lowercase t
u	117	Lowercase u
v	118	Lowercase v
w	119	Lowercase w
x	120	Lowercase x
y	121	Lowercase y
z	122	Lowercase z
{	123	Opening (left) brace
|	124	Vertical line
}	125	Closing (right) brace
~	126	Tilde
	127	Delete