DEALING WITH ERRORS IN THE MOTION OF A VEHICULAR ROBOT IN 2-D

by

Gordon C. Fossum

Memorandum No. UCB/ERL M81/35

29 May 1981

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Dealing With Errors in the Motion of a Vehicular Robot in 2-D

29 May 1981

by Gordon C. Fossum

Abstract

A software tool is described which simulates the motion of a special-purpose vehicular robot. This robot executes rotation and straight-line translations only, and is designed to operate without external assistance. The simulation incorporates the effects of seven different errors, all acting simultaneously, and of any one of four choices of compass aiding.

Introduction

Much research in the field of vehicular technology has been devoted to the development and analysis of systems designed to "get you from here to there." The vast majority of this research, however, has relied on the assistance of external inputs along the way, such as markers in the floor (Ref[1]), radio signposts (Ref[2]), or frequent recalibration by human operators or by a central computer with knowledge of likely paths (Ref[3]). (More references may be found at the end of Ref[2].)

The underlying assumption in such research is that errors in unaided systems grow too quickly for such systems to be useful for most applications. Unfortunately, data to support this assumption is not readily available. This paper should simplify efforts to provide such data.

The purpose of this paper is to examine the sources of errors in the motion of an unaided vehicular robot, describe their effects, and present a software tool, which is written in Pascal. This tool is a simulator, which, when given an input file consisting of (1) the type of compass guidance desired, (2) upper bounds on the errors, and (3) a list of ordered pairs of real numbers, (each pair representing a rotation command and a translation command, so that a,b means "rotate *a* degrees and move *b* meters"), will produce, as output, ordered pairs of real numbers with sufficiently fine granularity to closely approximate the path taken by the robot. The robot is presumed to be circular in top view with the center of mass at the geometric center, to simplify calculations later.

A more detailed and complete description of the software, as well as the results of its application to a chosen problem will follow the description of the errors.

Errors

There are seven error sources considered in this paper. They are treated as being independent of each other (such that changing the "severity" of any one of them has no effect on the severity of the others). They are examined singly (presuming that only one of them at a time has a non-zero value) to determine the "*error area*" each one can spawn. The errors are:

ABSOLUTE ROTATIONAL SKID (ABSSKID)

RELATIVE ROTATIONAL SKID (RELSKID)

ABSOLUTE ANGULAR ERROR (ABSANGL)

RELATIVE ANGULAR ERROR (RELANGL)

ABSOLUTE PATH LENGTH ERROR (ABSPATH)

RELATIVE PATH LENGTH ERROR (RELPATH)

PATH NON-LINEARITY ERROR (MINCURV)

The words in parentheses are the variable names for the maximum values of these errors in the Pascal program.

In all figures referred to in this section, the "correct" translational motion of the robot is toward the top of the page, which is to say that whatever the original orientation and angle of rotation prior to the translational motion, the "clever cameraperson" chose the appropriate angle to shoot from. Further-more, temporary *short* variable names are introduced, for the SOLE purpose of reducing the wordiness of the figures. In the figures, "t" (short for "theta") represents the commanded rotation (in degrees) and "d" represents the commanded translation (in meters).

1. ABSOLUTE ROTATIONAL SKID. Temporary name=e1; units=meters.

The first maneuver in the execution of the instruction is a rotation. In any real machine, the act of starting and stopping the rotation will cause the center of mass to move. The variable e1 represents the maximum (radial) distance it can "skid" due to starting and stopping, independent of the magnitude of the rotation. If this error acts alone, the robot, upon completion of its entire maneuver would stop somewhere in a circle of radius e1, centered on its intended destination (see Figure A).

2. RELATIVE ROTATIONAL SKID. Temporary name=k1; units=meters/deg.

Some skidding is caused by the rotation itself, and will be proportional to the magnitude of the rotation (unlike the previous error). The variable k1 is the maximum constant of proportionality which can be encountered. This error, acting alone, will cause the robot to stop inside a circle of radius $(k1)(|t|)$ centered on the intended destination (see Figure B).

3. ABSOLUTE ANGULAR ERROR. Temporary name=e2; units=degrees.

The act of starting and stopping the rotation will cause some error in the heading of the robot relative to what its heading should be. Roundoff error in calculating the rotation angle will also be included here. The variable e2 bounds these errors. Acting alone, e2 causes the robot to stop on (not inside) the circle of radius d, centered on the starting point, such that it is no more than e2 degrees of arc away from its intended destination (see Figure C).

4. RELATIVE ANGULAR ERROR. Temporary name=k2 (unitless).

Some inaccuracy is to be expected somewhere in the linkage of the rotational mechanism. The variable k2 bounds this proportional inaccuracy. This error causes the robot to stop on the same circle as that of the previous error, such that it is no more than (k2)(|t|) degrees of arc away from its intended destination (see Figure D).

5. ABSOLUTE PATH LENGTH ERROR. Temporary name=e3; units=meters.

The act of starting and stopping the linear motion will cause some error along the direction of motion. Roundoff error in calculating the path length will also be included here. The variable e3 bounds these errors. Acting alone, e3 causes the robot to stop on the line connecting the starting and stopping points within e3 meters of the intended stopping point (see Figure E).

6. RELATIVE PATH LENGTH ERROR. Temporary name=k3 (unitless).

Some inaccuracy will exist in the linkages of the translational mechanism. This error, proportional to the path length d, is bounded by the constant of proportionality k3, so that the robot will stop on the same line as that of the previous error within (k3)(d) of the intended stopping point (see Figure F).

7. PATH NON-LINEARITY ERROR. Temporary name=c; units=meters.

A variety of factors will cause the robot to follow some path which is not, in fact, linear. The variable c is the radius of the circle which the robot would trace if it exhibited worst case behavior. The reciprocal of c is the maximum curvature which the robot's path can exhibit. Thus, if the value of c were 20.0, the robot could move in a circle of radius 20 meters, and have a maximum curvature of 0.05. (Throughout the following two paragraphs, refer to Figure G.)

This error is the tough one to analyze, because it allows the robot such freedom. The robot's path can be visualized as a rope of length d, immovably anchored at one end (the starting point) and projecting along some flat floor. The rope is only slightly flexible, such that it can be bent into a curve of radius c at any point, but no more.

The job is to describe the closed curve in 2-space such that points on and inside the curve represent possible positions of the free end of the rope, and points outside the curve cannot be reached by the free end of the rope. Three extreme points can be shown to be on the curve immediately. The intended destination is on the curve, because, in some sense, you can't go "farther" than that point from the starting point. Two other points on the curve are those arrived at by proceeding "hard to port" or "hard to starboard" from the starting point, for the entire path length d. The "hard to port" point is located (are you ready?) on a circle of radius c whose center is at a distance c from the starting point, to the "left", along the line containing the starting point which is perpendicular to the intended path of the robot such that the length of the arc along this circle from the starting point up to the "hard to port" point is d. The "hard to starboard" point is found in an analogous fashion.

These three points can be connected by a curve which may be viewed as being obtained by swinging the rope from "hard to port" through the intended destination to the "hard to starboard" point, keeping it as taut as possibl• at all times. A little reflection leads one to conclude that a snapshot of the rope at an arbitrary point on this journey will show the rope starting on a circular arc to some point and then continuing as a straight line (see Figure H). This curve forms the outer boundary to the error area, as there is no way for the robot to go beyond it (if all other errors are zero).

Another curve is found by "pushing" the rope (perhaps by attaching a spring to connect the two ends of the rope) so as to minimize the radial distance between the two ends. A rope compressed in this fashion will assume an S shape built of two circular arcs, each of radius c, as no other permitted configuration is as "short" in radial length. (At this point, however, a proof of this conjecture remains elusive.) This curve is not symmetric, however, and we must superimpose two versions of it (one arrived at by "peeling" the rope from left to right, the other from right to left) and take, as our inner boundary to the error area, those portions "closest" to the starting point (see Figure I). These curves, then, define the area caused by the PATH NON-LINEARITY ERROR.

Error Composition

The absolute and relative angular errors can be combined, resulting in an arc which is wider than either of them. Similarly, the absolute and relative path length errors can be combined. If all four of these errors are considered together, the resulting error area resembles the swath of a windshield wiper (see Figure J). This error area represents the final answer if we could assume that the robot really could "turn on a dime" and really did travel in straight lines.

For the most general picture, though, it's best to start with the path non-linearity error area (Figure I), and "fold in the other ingredients carefully."

First, add the path length errors in by considering that the path non-linearity error presumes a path of exact length. If we substitute for the "correct" path length the longest path length permitted by the two path length errors (taken together), we get a slightly larger path non-linearity error area which is positioned slightly farther away from the starting point. An analogous, smaller, closer area is derived from the shortest path length permitted by the path length errors.

Now, recall that the "hard to port" and "hard to starboard" points were on the circles of tightest curvature (smallest radius) that the robot could travel, but the curvature itself is independent of (and therefore constant throughout) the just-completed construction. Thus, as the path length varies from longest to shortest, these two points sweep out circular arcs, which, taken in union with the outer (upper) curve of the larger path non-linearity error area and the inner (lower) curves of the smaller path non-linearity error area, define the new composite error area which specifies where the robot could be if paths of inexact length and imperfect linearity were permitted, but all rotations were presumed to be executed perfectly (see Figure K).

The angular errors can next be incorporated into this composite error area easily by just rotating the entire area about the starting point clockwise and counter-clockwise to the maximum angle permitted by the angular errors taken together (-e2-(k2)(|t|) to +e2+(k2)(|t|)) (see Figure L).

The final composite error area is achieved by incorporating the "skid" errors. This is accomplished by stating simply that the new error area consists of all points which are within a distance of (e1+(k1)(|t|)) of some point on the previous error area. This production can be visualized as painting a border of this width, with rounded corners, around the previous error area (see Figure M).

Compasses

There are two things to consider here. First, compass error and tolerance, and second, how a compass can help keep errors small.

Any compass, magnetic or gyroscopic, will have some small error in its own function. One assumes that this error is significantly smaller than the angular errors described above, as this is necessary to justify using the compass. In some sense, the compass (and associated circuitry) "reacts" when it senses that the heading of the robot is, at some point, different from the heading it "should" have, by more than some tolerance. The tolerance programmed into the compass should be larger than the error inherent in the compass, to avoid anomalous behavior.

The circuitry that receives information supplied by the compass might use it to effect "midcourse corrections" by "locking in" the heading of the robot at the beginning of each translation, and executing internally generated rotation commands during the course of that translation whenever the heading strays from the locked-in "standard" by more than the compass tolerance.

The circuitry could, on the other hand, store a running sum of the rotation commands being executed, and thus always have a record of what the heading *really should be;* thus, after each rotation is executed, the compass circuitry could effect small corrections *before* the translation begins.

The abstract of this paper mentioned four choices of inertial aiding. The first choice is no compass (no aiding at all). The second choice is the compass of the first kind above. This is called "Dumb Compass 1" on the graphs of the simulations contained in this paper. The third choice is the compass of the second kind above. This is called "Dumb Compass 2". The fourth choice is both "Dumb Compasses" working together. This is called "Smart Compass" on the graphs.

The Simulation Program: An Explanation

As described earlier, the simulation program is expected to output the coordinates of the path taken by the robot in executing the commands input to it. In fact, it does more than this. The program (a listing of which follows the figures at the end of the paper) is entitled "TARGET" because it is designed to stop a given simulation without executing all commands if the robot ever "hits" a target, whose coordinates are selectable by changing the values of "xtarget" and "ytarget" in the *const* section of the program. If it is desired that all simulations run to completion, the coordinates of the target may be set to some distant point.

There are four procedures in the program. They are "buildlist", "move", "xeqcmd" and the main procedure. An explanation of each procedure follows:

"*buildlist*"

This simple procedure creates a linked list out of the input commands, so that they can be used multiple times by the main program.

"*move*"

This procedure takes, as input, the magnitude and direction of a desired incremental move of the robot, calculates the new x,y coordinates of the robot, checks to see if any point along this incremental move is within a distance equal to the radius of the robot of the target, and prints out the new position of the robot.

"*xeqcmd*"

Short for "execute command", this procedure contains the mathematics required to calculate the effects of the seven errors on each command to be executed. Different values for the errors are chosen randomly for each command execution under the assumption that all values within the prescribed limits are equally likely. That is, if x is the maximum value for a given error, it is assumed that the probability curve for that error is a uniform distribution in the interval [-x,x]. The size of the incremental moves for the command are determined as a function of MINCURV and the path length of the command, and the required incremental moves are calculated and executed, by calling the "move" procedure; compass corrections are incorporated when needed, if the compass level (COMPLVL) indicates that the robot has sufficient "smarts".

A bit more explanation is in order with regard to the random selection of error values referred to above. Values for the first six errors are chosen once per command, but values for the path non-linearity error are chosen randomly (with some bias) *once for each incremental move within a command.* To understand the "bias" in these incremental errors, it is necessary to investigate what causes non-linearity.

The contributors to non-linearity fall into two classes. First, there are those that are effectively constant in time, such as wheels with different radii, which would tend to cause the robot to move in a fairly well-behaved circle. Then there are those that are truly random in nature, such as small irregularities in the surface on which the robot travels. Any combination of these two classes of errors may be selected to yield a series of path non-linearities, all bounded by MINCURV, and all related to their predecessors and successors to some degree.

One might expect that a combination with a heavy weighting on surface irregularities, rather than wheel irregularities, would yield results more closely clustered around the correct destination than would a combination biased more toward the approximately constant inaccuracies of poorly matched wheels. This suspicion was borne out by some simulations, two of which are shown in Figures N1 and N2. The variable SMOOTHWT can be visualized as a measure of the smoothness of the floor: thus, a high SMOOTHWT means that the bulk of the error was in the wheels. These figures show clearly that simulations with high SMOOTHWT are more evenly distributed across the entire "error area". For this reason, SMOOTHWT=50.0 was chosen for the simulation program.

"*target*"

The main program reads the input parameters, calls "buildlist", then runs through the linked list of commands *n* times, where *n* is specified in the *const* section. The "hit percentage" is then printed in such a way that it will appear on the graph near the origin.

Some System-Dependent Details

The "random" function returns pseudo-random numbers in the interval (0,1). Its randomness was checked by plotting 50,000 points at random in the unit square. No patterns were discernable in the resulting graph.

Each line output by the program consist of two real numbers, possibly followed by a comma. The numbers generated by the program are fed into a system routine called "graph" which generates a connected line as long as it sees just two real numbers (and nothing else) on each line. When something else is on a line, the graph is broken at that point, and the point is "labelled" with the "something". A comma was the most innocuous label available, so every simulation is therefore plotted with a comma marking its termination.

Exercising the Simulator: some algorithms

In order to test the capabilities of the simulator, the following problem was posed. Imagine that a robot of radius 0.5 meters is required to travel from (0 meters, 0 meters) with initial heading along the positive y-axis to (15 meters, 15 meters), and hit the target located there. If the errors are small enough, a single command would suffice. However, if the errors are realistic, some reasonably intelligent algorithm might be required to maximize the likelihood of a hit.

A total of six algorithms were developed and tested under all four levels of compass aiding, and using two different collections of error bounds; one emphasizing bad angular errors, and the other emphasizing bad path length errors.

The first four algorithms are based on calculating the error area around the target for each given set of error bounds, enclosing this error area with the smallest rectangle possible, and generating commands which cause the robot to "cover" that rectangular area. (Of course, the robot will *not* cover the area perfectly because of the cumulative nature of the errors involved in multiple commands. The purpose of this exercise is *not* to decide what the *best* algorithm is, but to show how the simulator can compare the performance of several given algorithms.) The four "rectangle-covering" algorithms are called "long", "short", "side", and "spiral", and examples, showing how they would look if they worked perfectly, are contained in Figures P1 through P4, with the error area they are designed to cover shown in Figure P5. In the "long", "short", and "side" algorithms, the "zig-zag" movements are such that the robot progresses a distance equal to its diameter for every pair of commands executed. In the "spiral" algorithm, after each loop, the robot is similarly one diameter further away from the center than before. (It is very likely that different simulation results would be obtained if algorithms were used which caused slower or faster progress.)

The last two of the six algorithms mentioned are "random". These algorithms merely command the robot to move to (15 meters, 15 meters) and start moving randomly from there (totally random rotation commands, and random translation commands biased toward the diameter of the robot). The two algorithms differ in that the second random algorithm is programmed to execute four times as many random steps before giving up; in both algorithms, the number of random commands is a function of the shape and size of the rectangle surrounding the error area.

Five series of runs were generated. In the first, only ten iterations of each simulation were calculated (and graphed) to show visually what happens. Some of these are shown in Figures Q1 through Q30. (These figures appear in six groups of five figures each. Each group of five figures shows the performance of one of the six algorithms described. Within each group, the first four figures show the performance of that algorithm for each compass level, given *bad angular errors*, and the fifth uses "no compass" (the first compass level) only, assuming the scenario of *bad path length errors.)* (The actual error bounds employed in each of the two scenarios are shown in Figure R.) In the second through fifth series, respectively, 500, 650, 800, and 1000 iterations of each simulation were calculated (but not graphed) to yield the hit percentages for each. The average of these last four is shown in Figure S.

Conclusions: Shortcomings and Directions for Future Research

Even a cursory examination of Figure S shows that for the bad angle error ensemble, the Spiral algorithm outperforms all others, and intelligent planning *does* count for something, as the performance of the random algorithms shows. Further, the conclusion that bad angles are more troublesome than bad path lengths is probably warranted. However, compasses can improve performance in both cases (more so in the bad angle case, as one might expect). It is interesting to note that the "side" algorithm responds to the assistance of the "dumb" compasses differently than *all* of the other algorithms.

On a more fundamental level, some conclusions and warnings about the program itself, as opposed to the results of their use, are in order.

The program makes some bad assumptions. One is that the error values are really uniformly distributed. This may very well not be true. A second bad assumption is that the error values are completely independent from one command to the next. It would not be too difficult to correct this second assumption if only it were known to what degree each error should be correlated from one command to the next.

This simulator could be expanded to investigate the performance of robots programmed to try to cover *entire areas* with no outside assistance, rather than just trying to hit a single target. An internal array of points could be individually flagged during the "move" subroutine if and when each of them was approached closely enough by the robot.

Acknowledgments

My sincere gratitude goes to Manuel Blum, Professor, Computer Science Division, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, for his spirited interest and valuable guidance in this project.

This research was sponsored by the National Science Foundation, Grant MCS-79-03767-A01.

References

[1] Yoshikuni Okawa, "An Application of a Microcomputer to Dead Reckoning of an Electric Cart", *Conference Proceedings IECI Annual Conference*, pp 400-405, 1980.

[2] M. D. Kotzin and A. P. van den Heuvel, "Dead Reckoning Vehicle Location Using a Solid State Rate Gyro", *Conference Record IEEE Veh. Tech.*, Vol. 28, pp 169-172, 1978.

[3] Jon Myer, "VEPOL-A Vehicular Planimetric Dead-Reckoning Computer", *IEEE Trans. on Veh. Tech.*, Vol. VT-20, No. 2, pp 62-68, 1971.

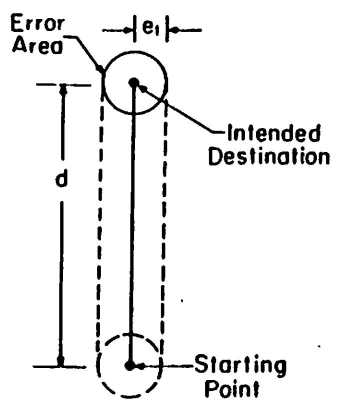

FIGURE A

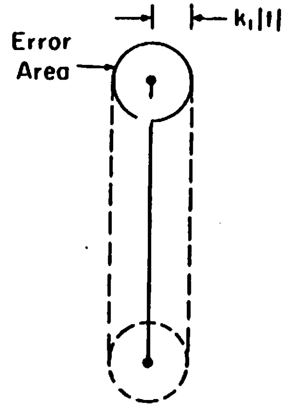

FIGURE B

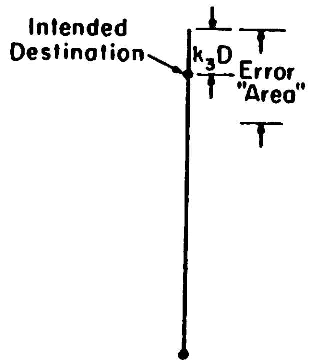

FIGURE F

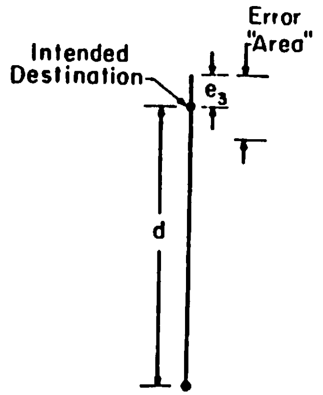

FIGURE E

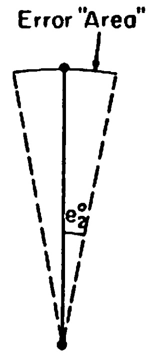

FIGURE C

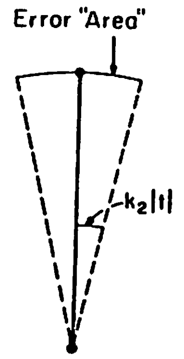

FIGURE D

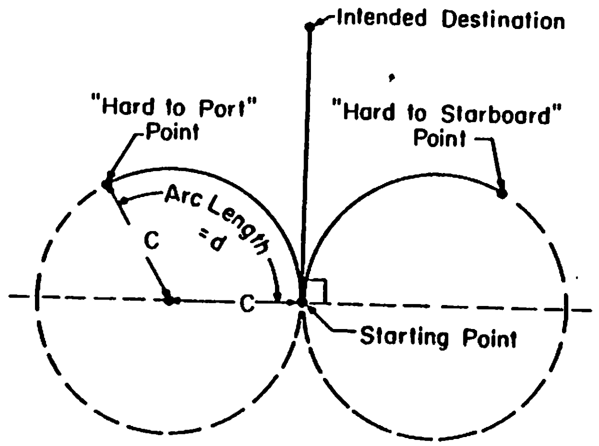

FIGURE G

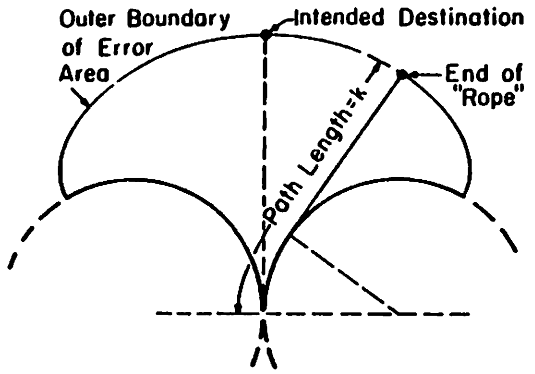

FIGURE H

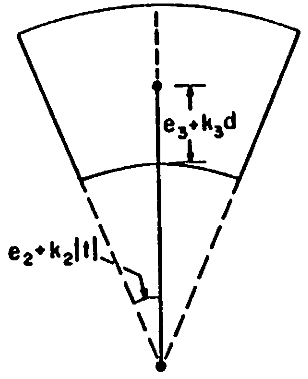

FIGURE J

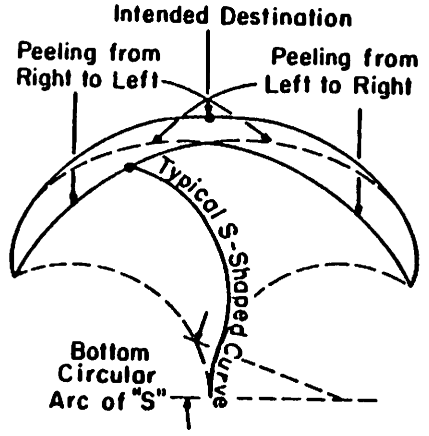
Intended Destination
Peeling from Right to Left
Peeling from Left to Right
Typical S-Shaped Curve
Bottom Circular Arc of "S"

FIGURE I

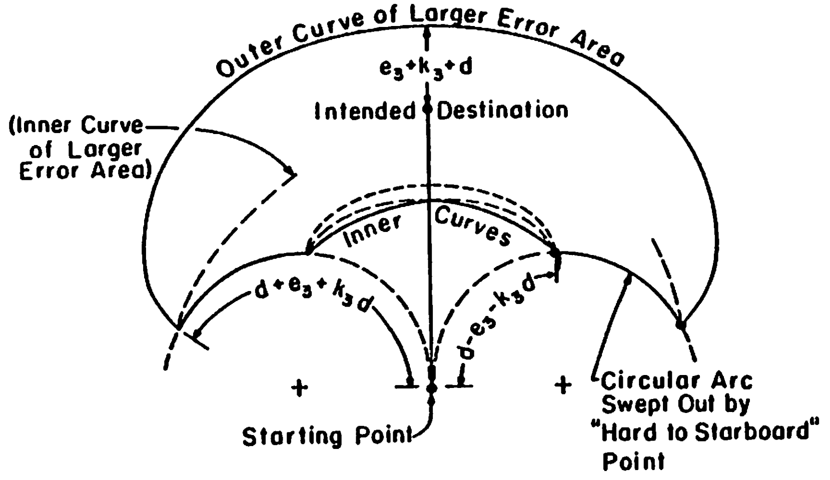
Outer Curve of Larger Error Area
e_3+k_3+d
Intended Destination
(Inner Curve of Larger Error Area)
Inner Curves
$d+e_3+k_3d$
$d-e_3-k_3d$
Starting Point
Circular Arc Swept Out by "Hard to Starboard" Point

FIGURE K

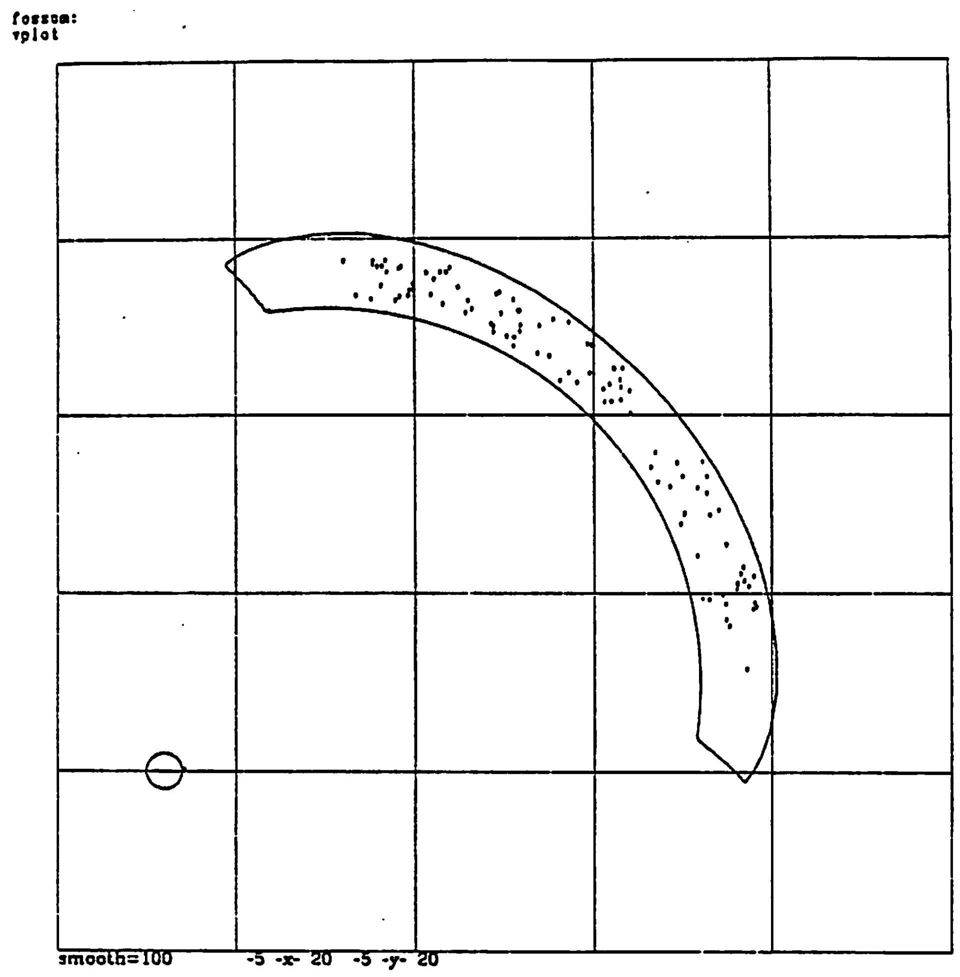

Figure N2
Example with High "SMOOTHWT"

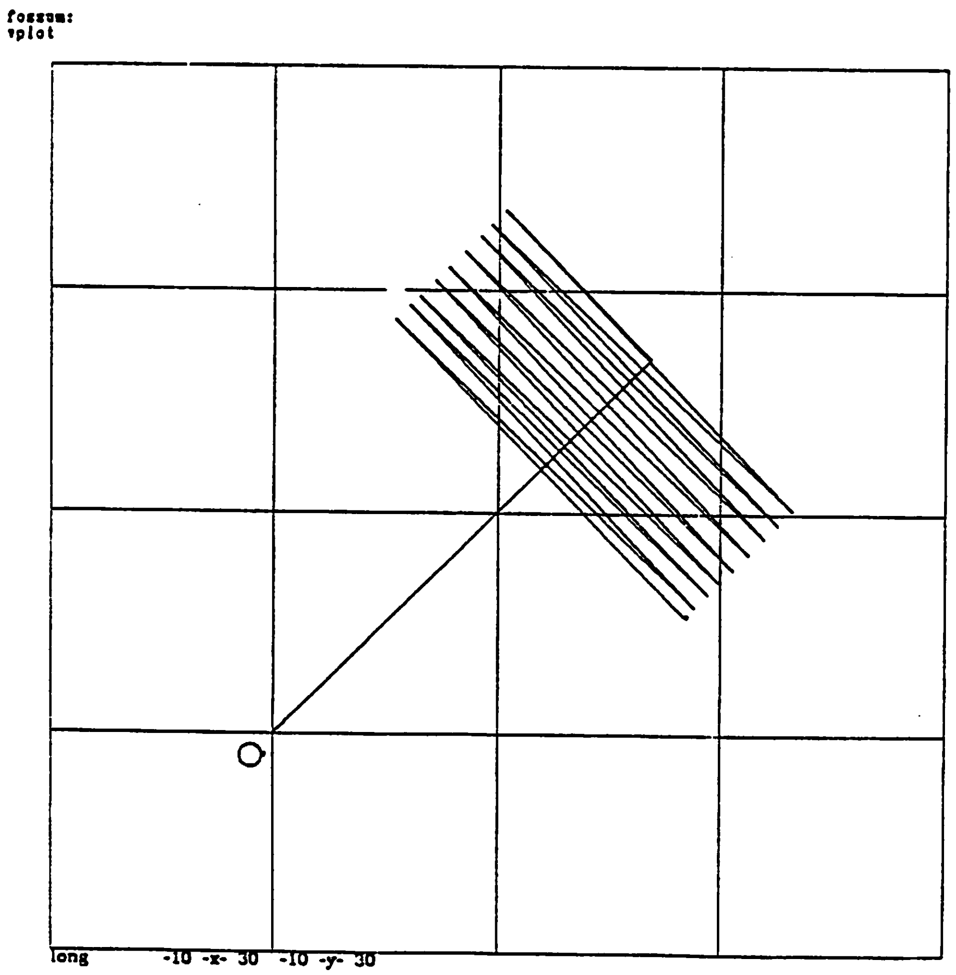

Figure P1
Long Algorithm

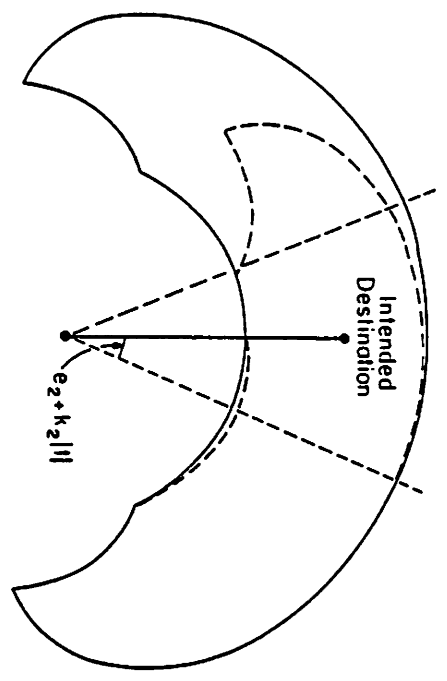

FIGURE L

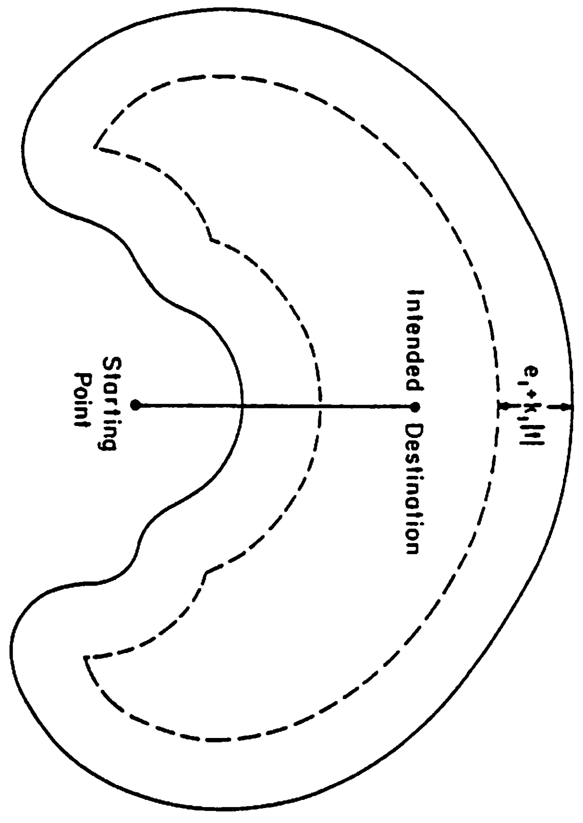

FIGURE M

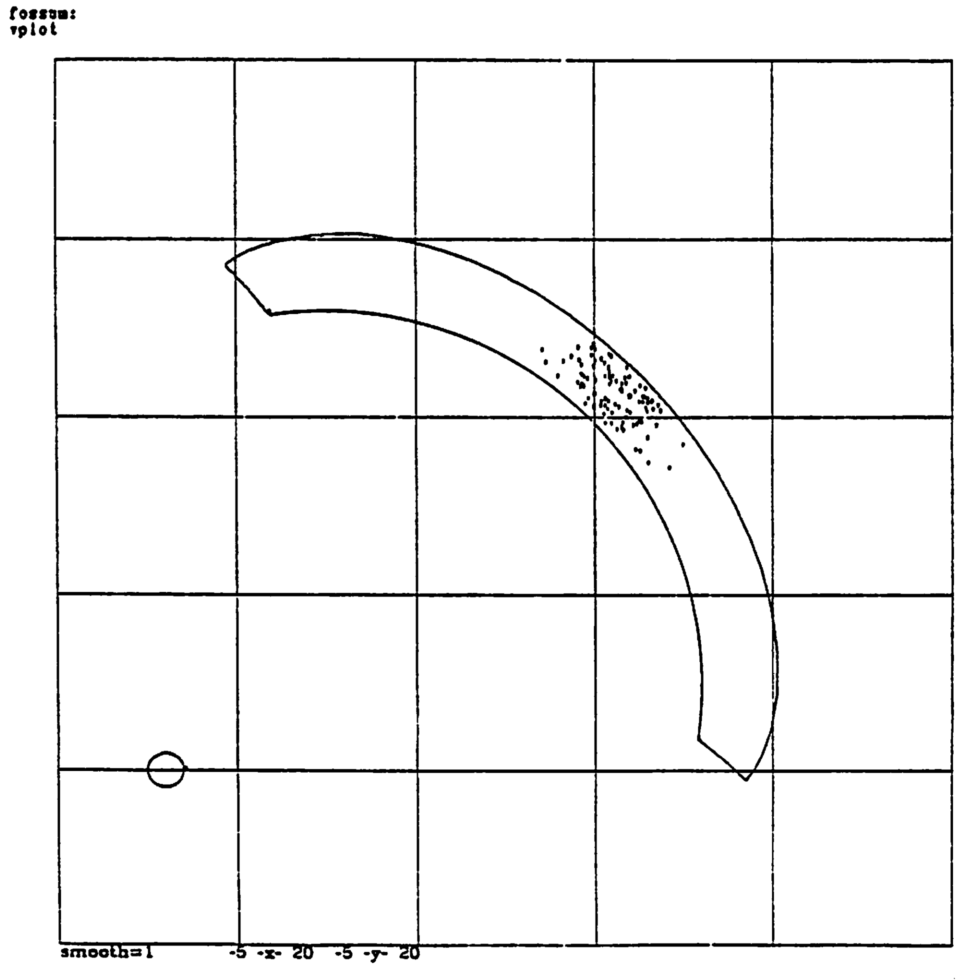

Figure N1
Example with Low "SMOOTHWT"

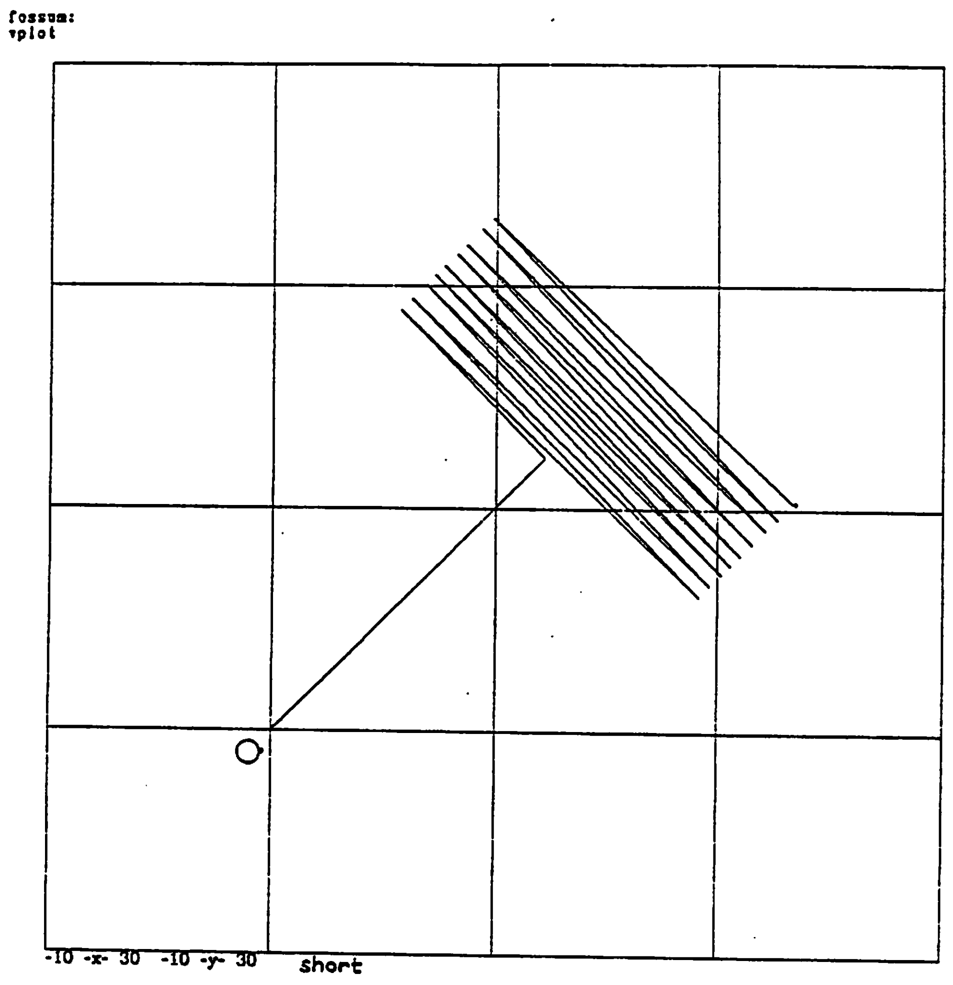

Figure ~~02~~ P2
Short Algorithm

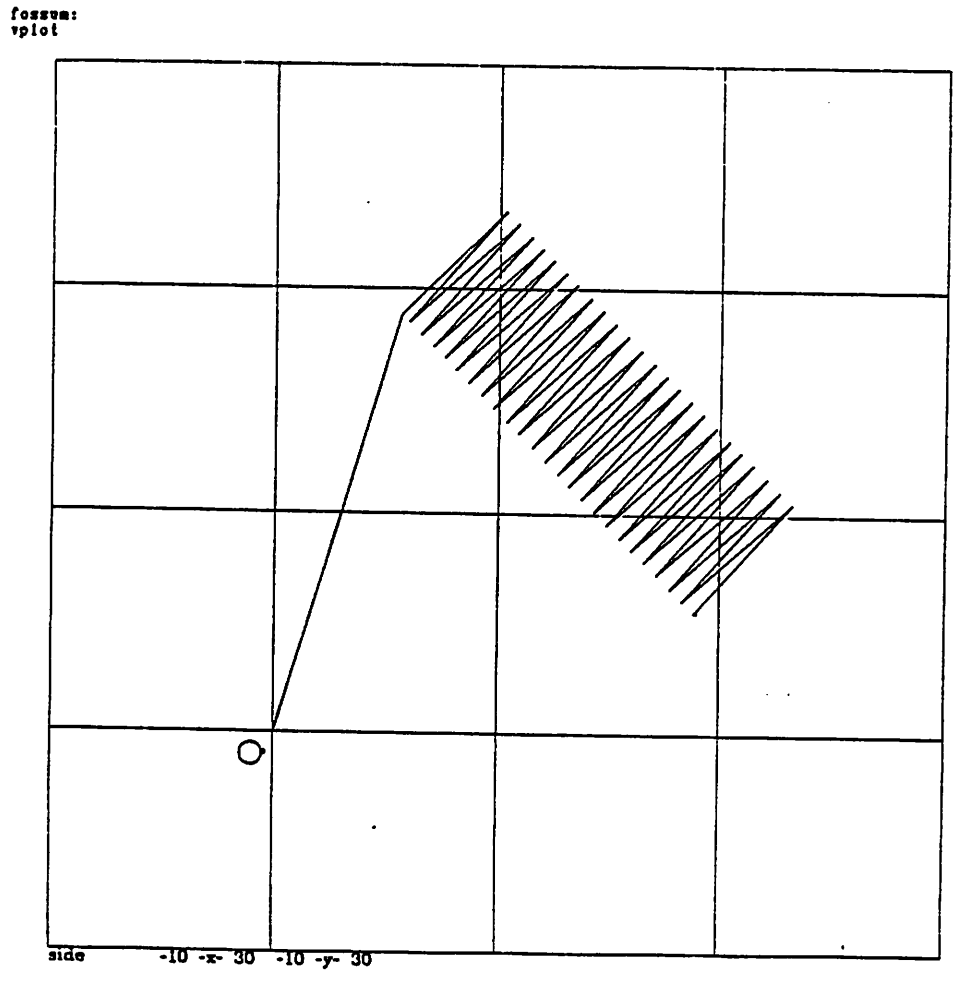

Figure P3
Side Algorithm

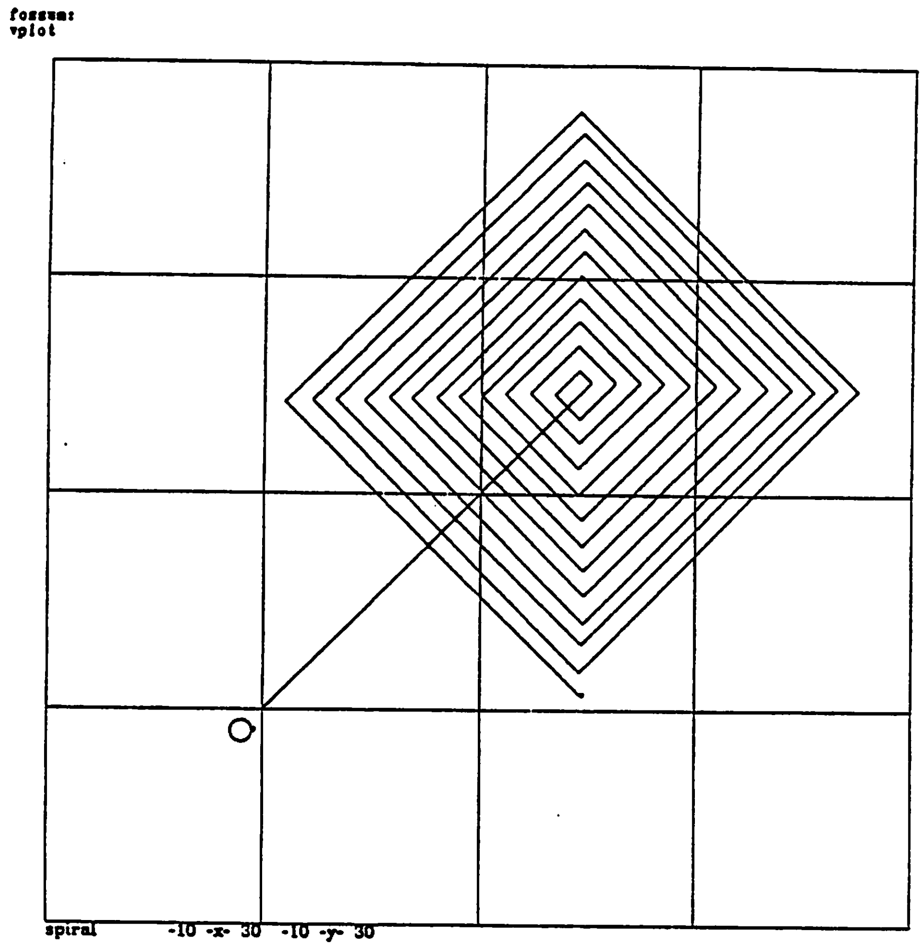

Figure P4
Spiral Algorithm

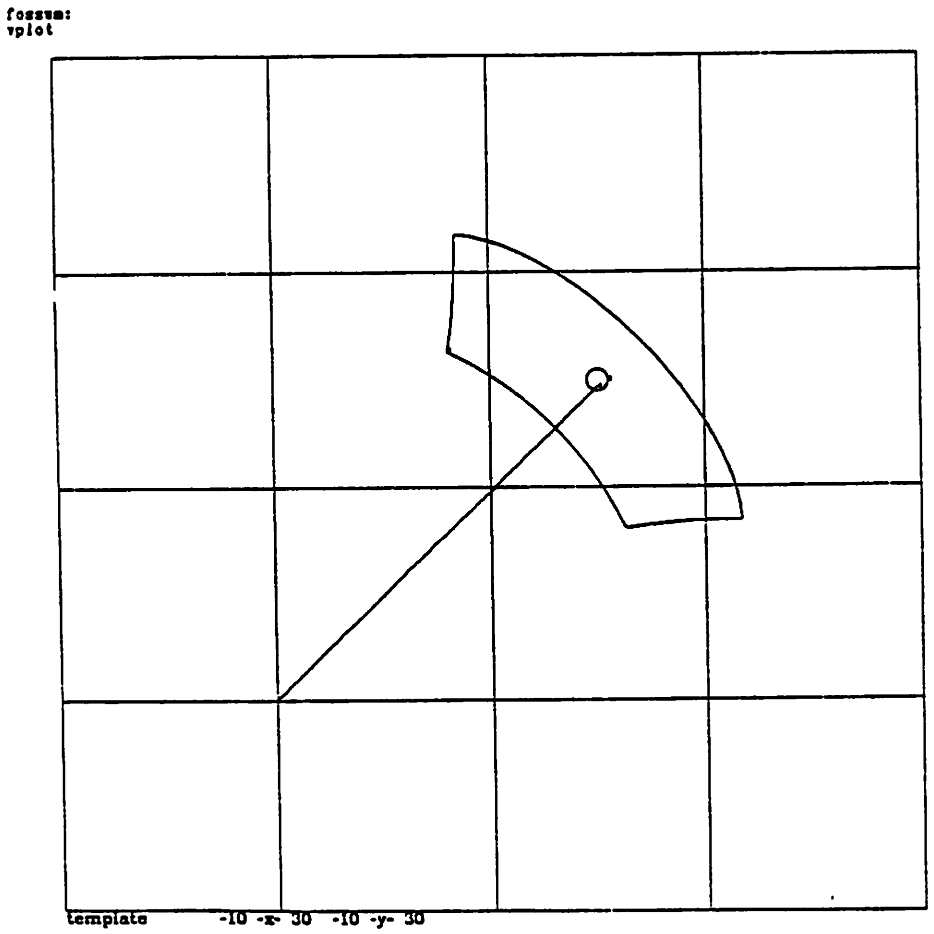

Figure P5
Error Area Covered

fossum:
vplot

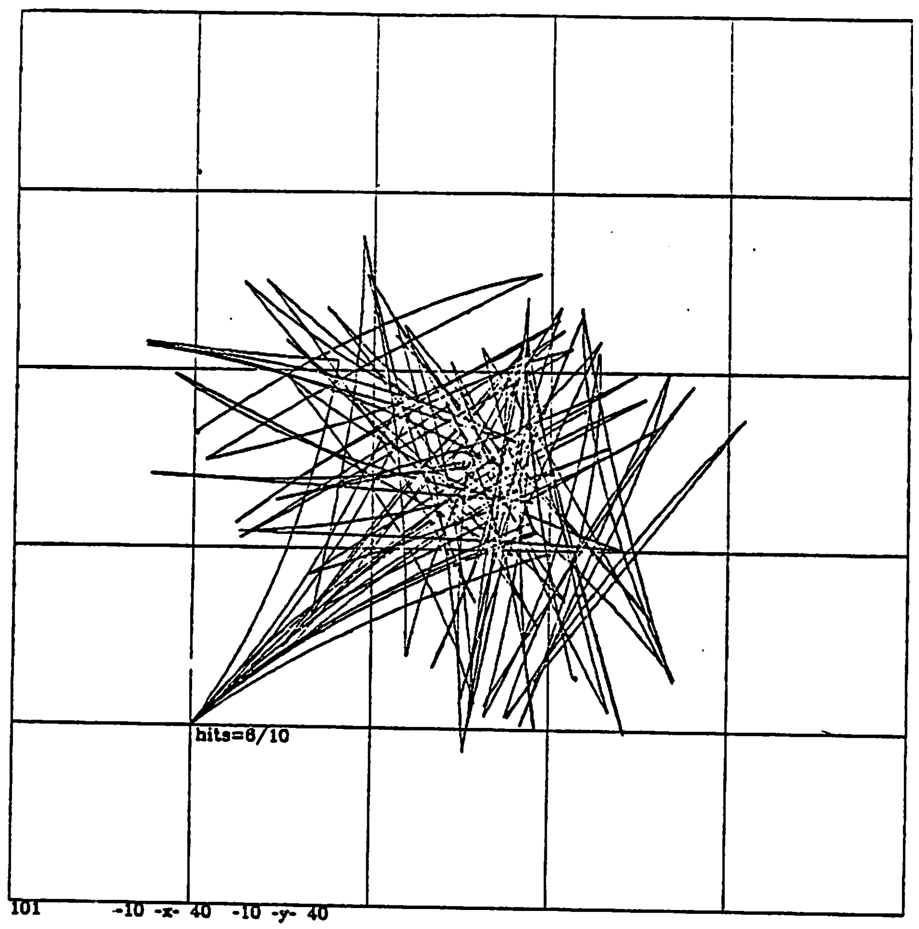

Figure Q1
Bad Angle Errors
Overshoot Algorithm
No Compass

fossum:
vplot

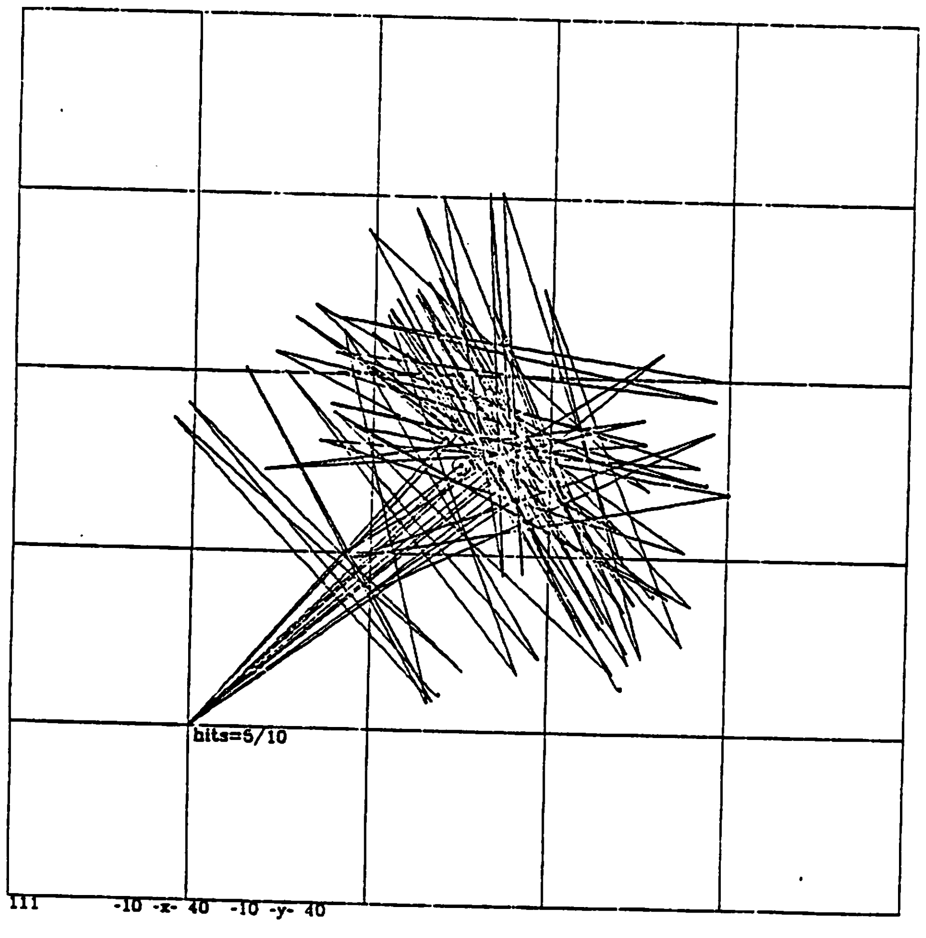

Figure Q2
Bad Angle Errors
Overshoot Algorithm
Dumb Compass 1

fossum:
vplot

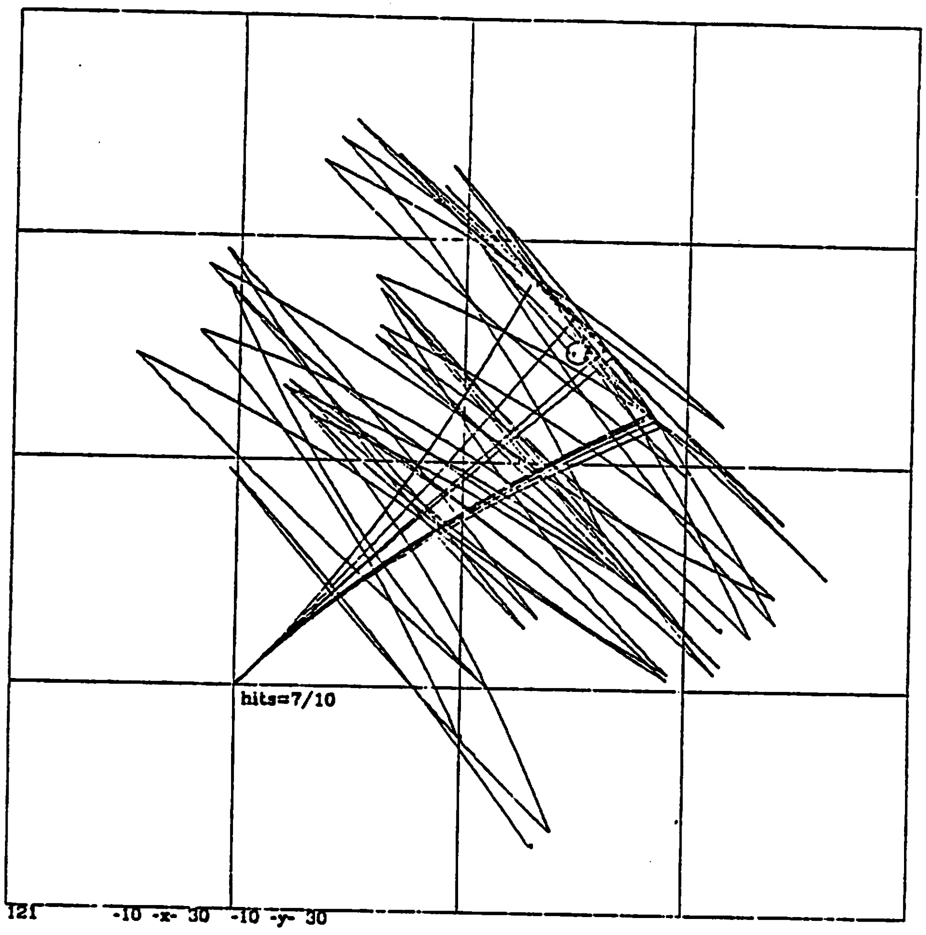

Figure Q3
Bad Angle Errors
Overshoot Algorithm
Dumb Compass 2

fossum:
vplot

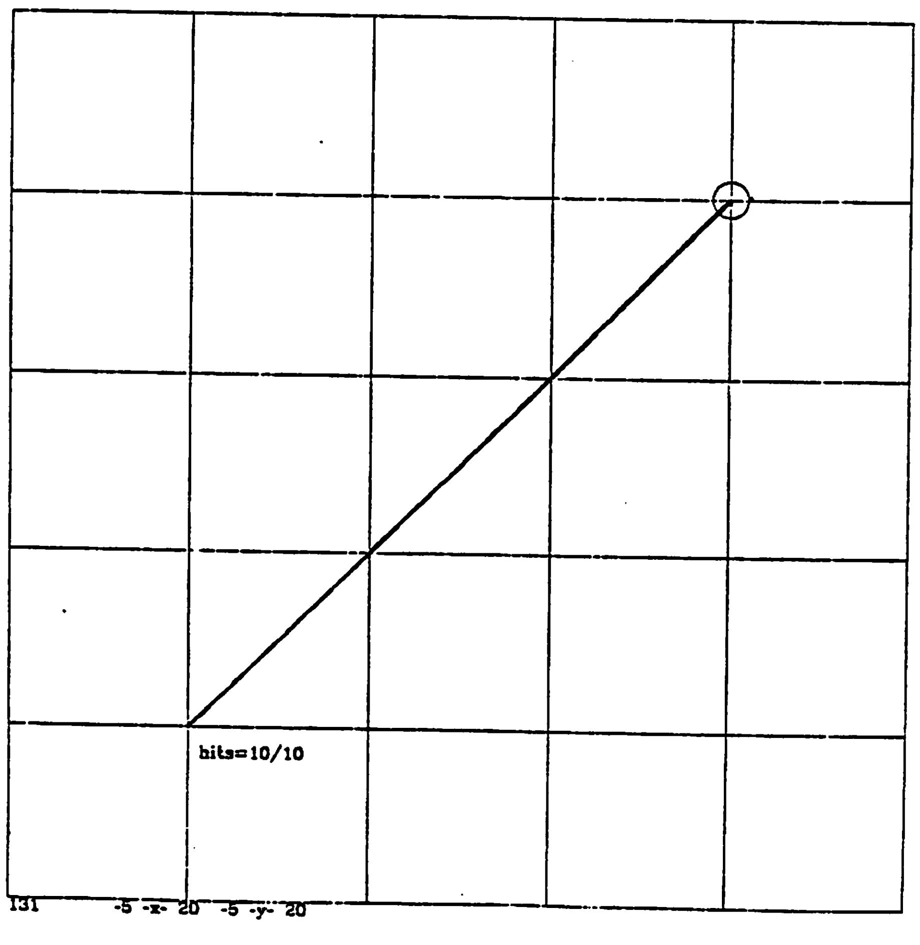

Figure Q4
Bad Angle Errors
Overshoot Algorithm
Smart Compass

fossum:
vplot

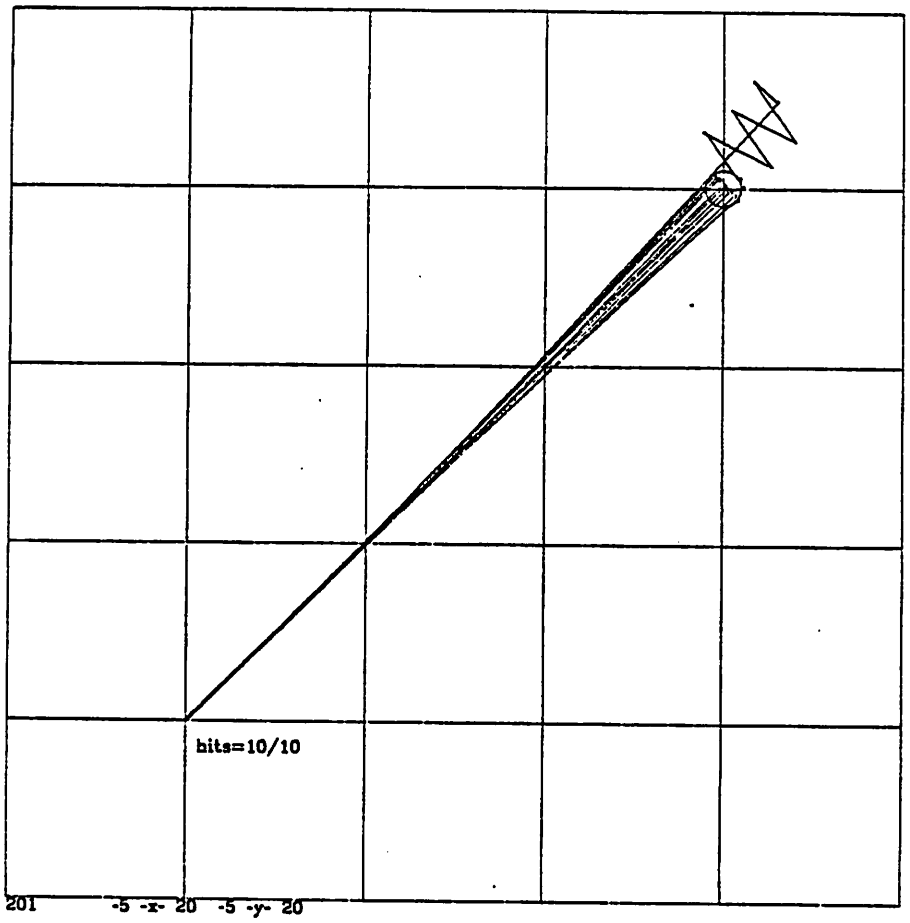

Figure Q5
Bad Path Errors
Overshoot Algorithm
No Compass

fossum:
vplot

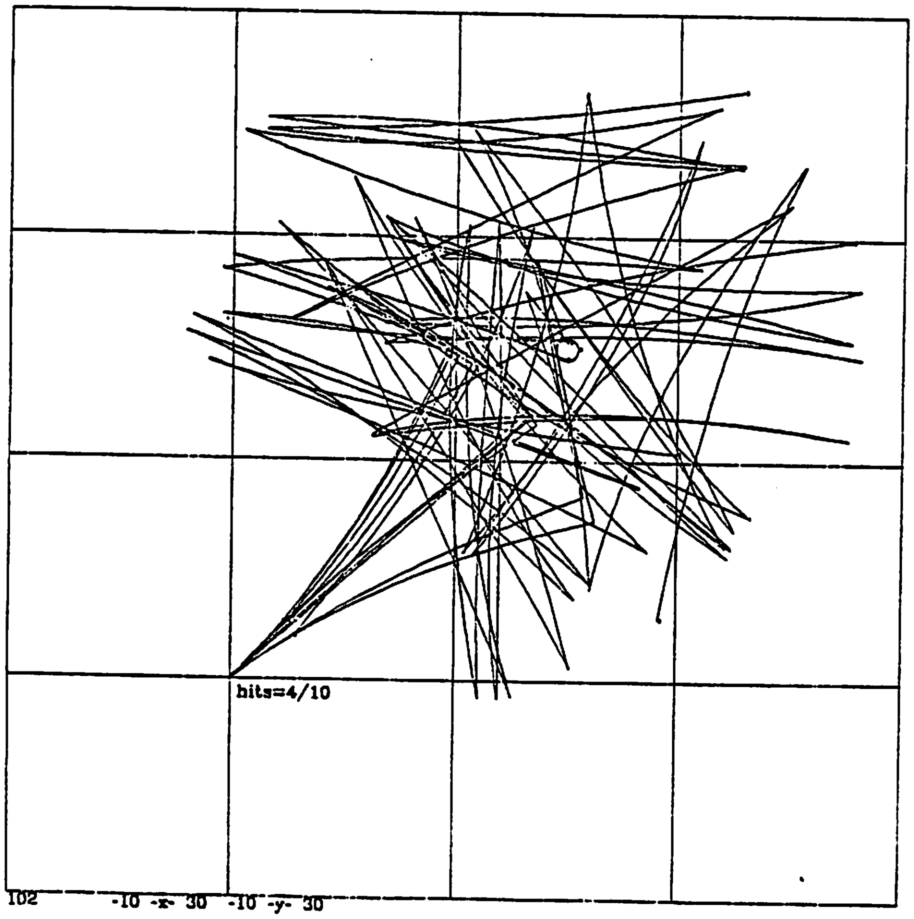

Figure Q6
Bad Angle Errors
Undershoot Algorithm
No Compass

fossum:
vplot

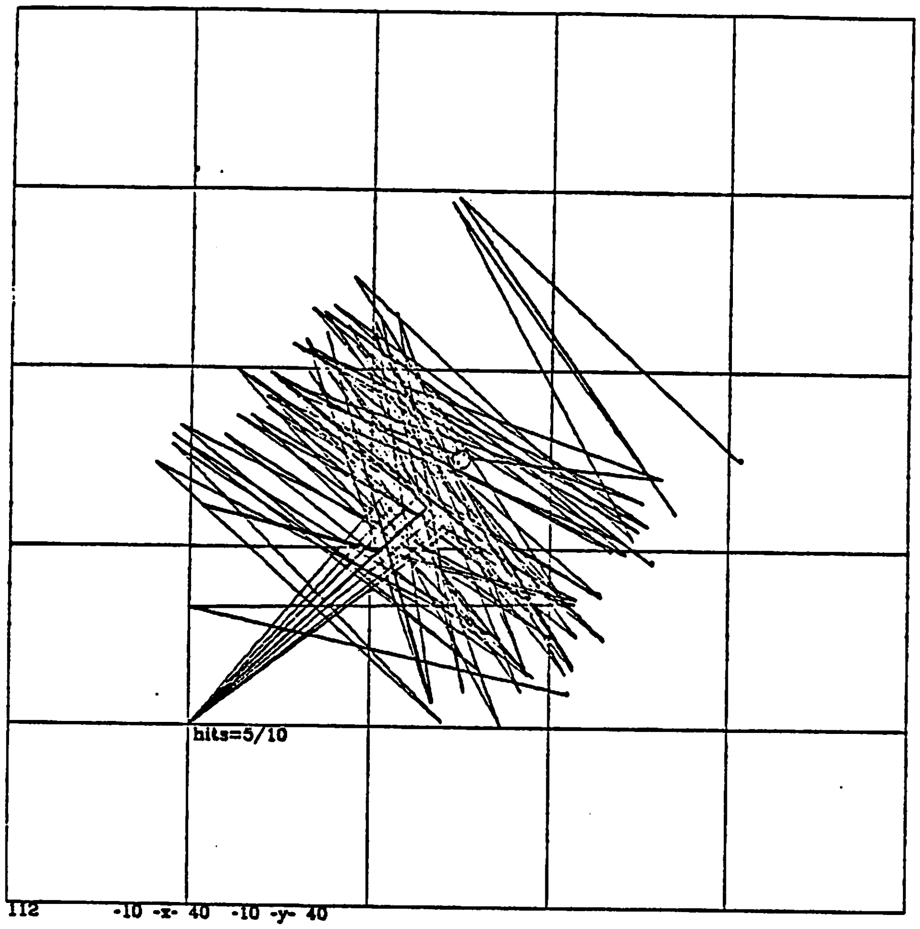

Figure Q7
Bad Angle Errors
Undershoot Algorithm
Dumb Compass 1

fossum:
vplot

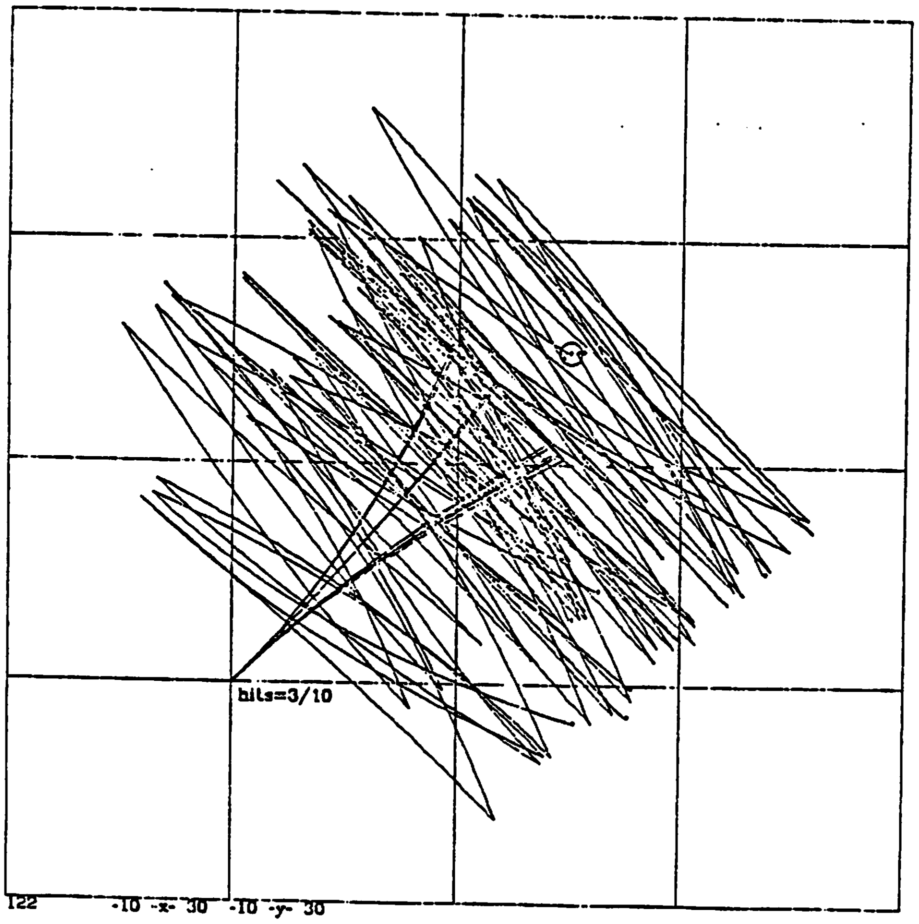

Figure Q8
Bad Angle Errors
Undershoot Algorithm
Dumb Compass 2

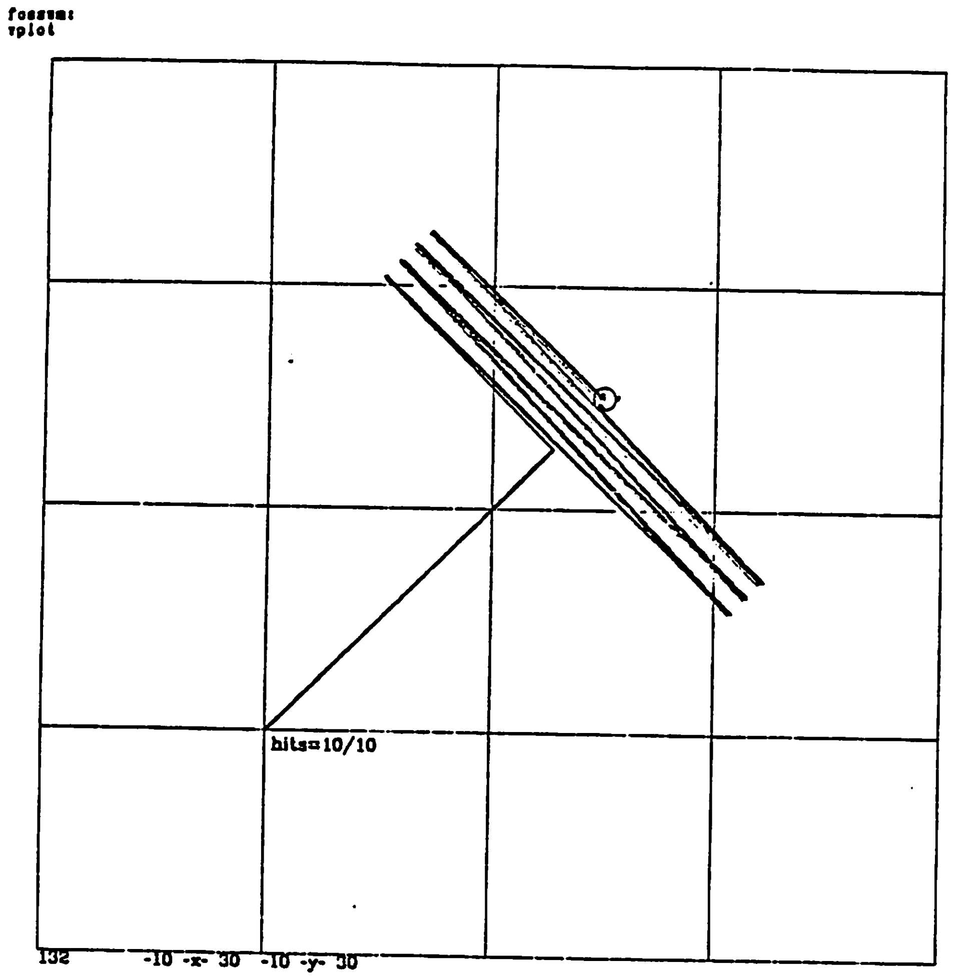

Figure Q9
Bad Angle Errors
Undershoot Algorithm
Smart Compass

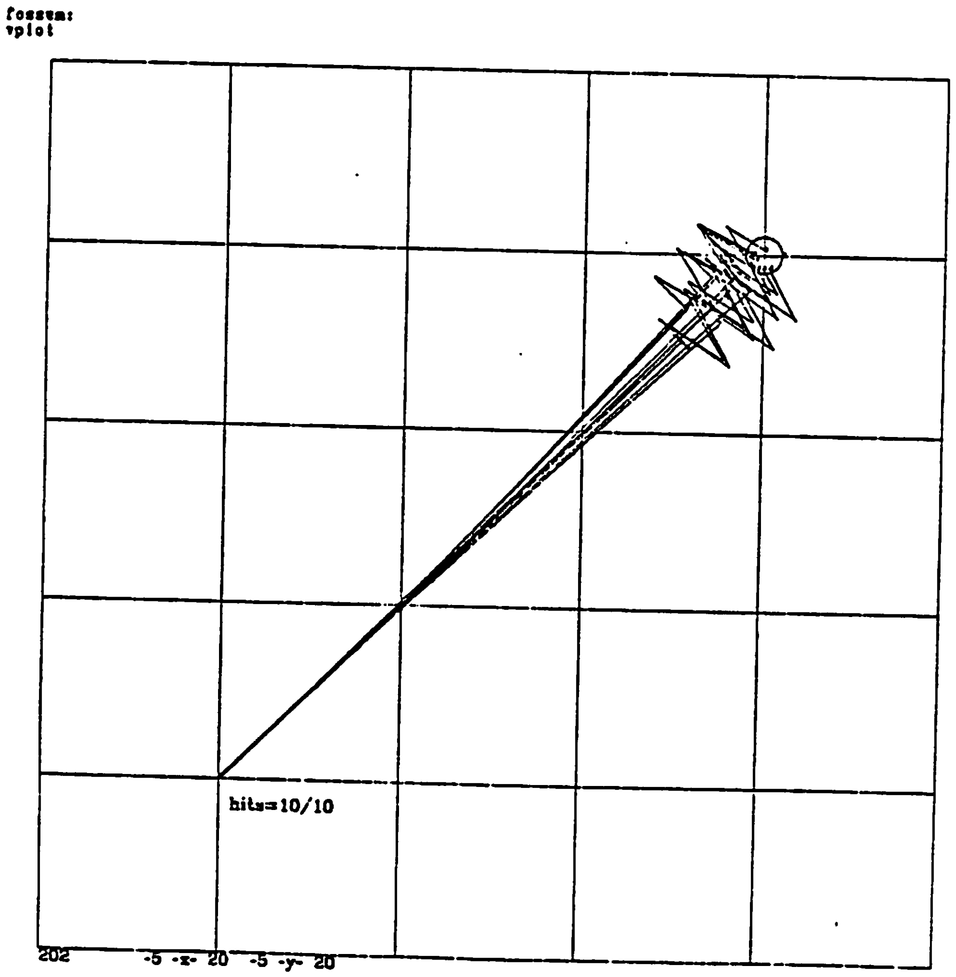

Figure Q10
Bad Path Errors
Undershoot Algorithm
No Compass

fossum:
vplot

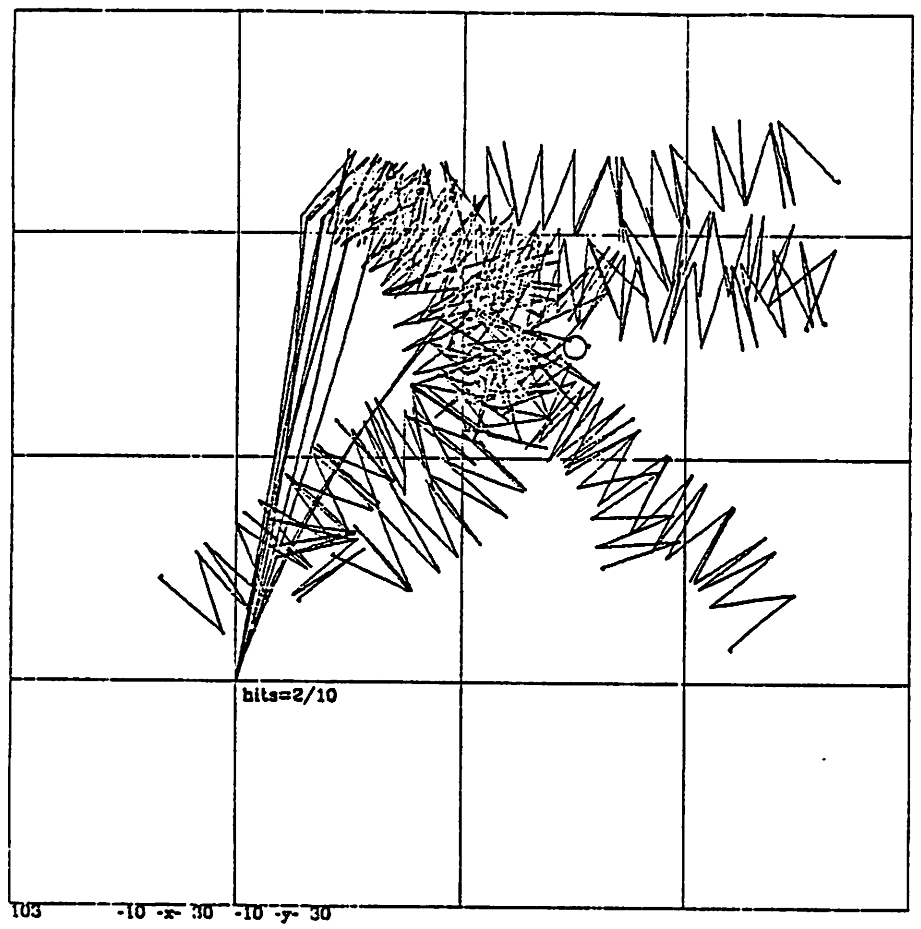

Figure Q11
Bad Angle Errors
Side Algorithm
No Compass

fossum:
vplot

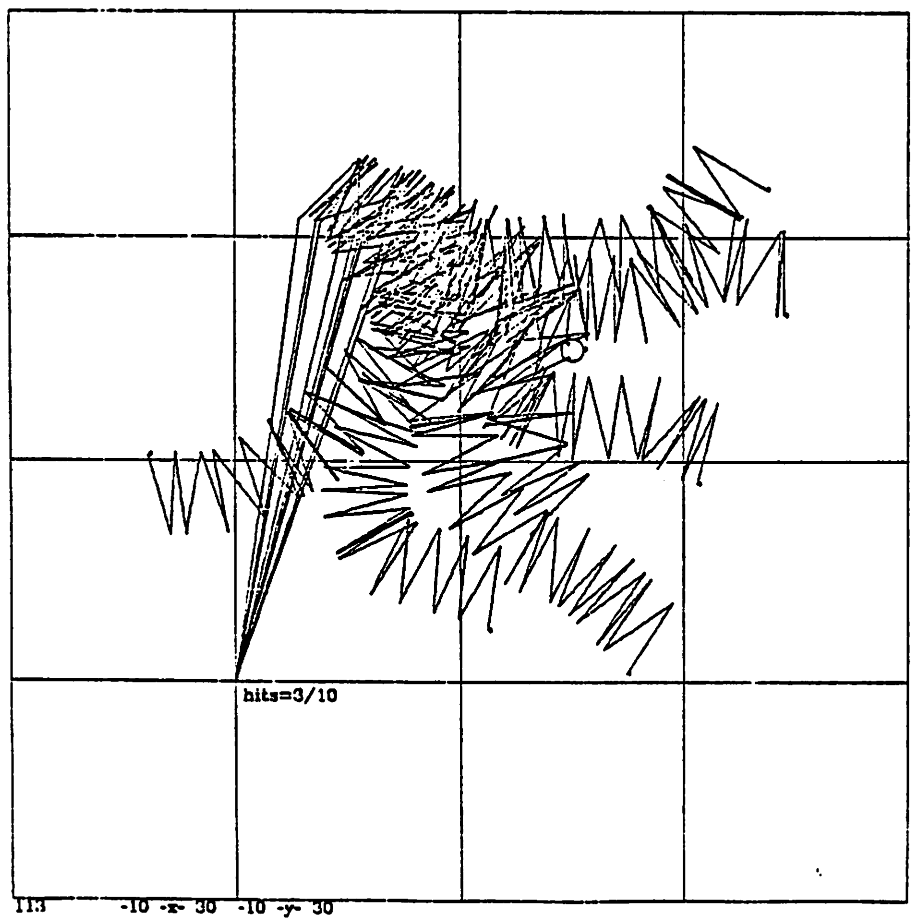

Figure Q12
Bad Angle Errors
Side Algorithm
Dumb Compass 1

fossum:
vplot

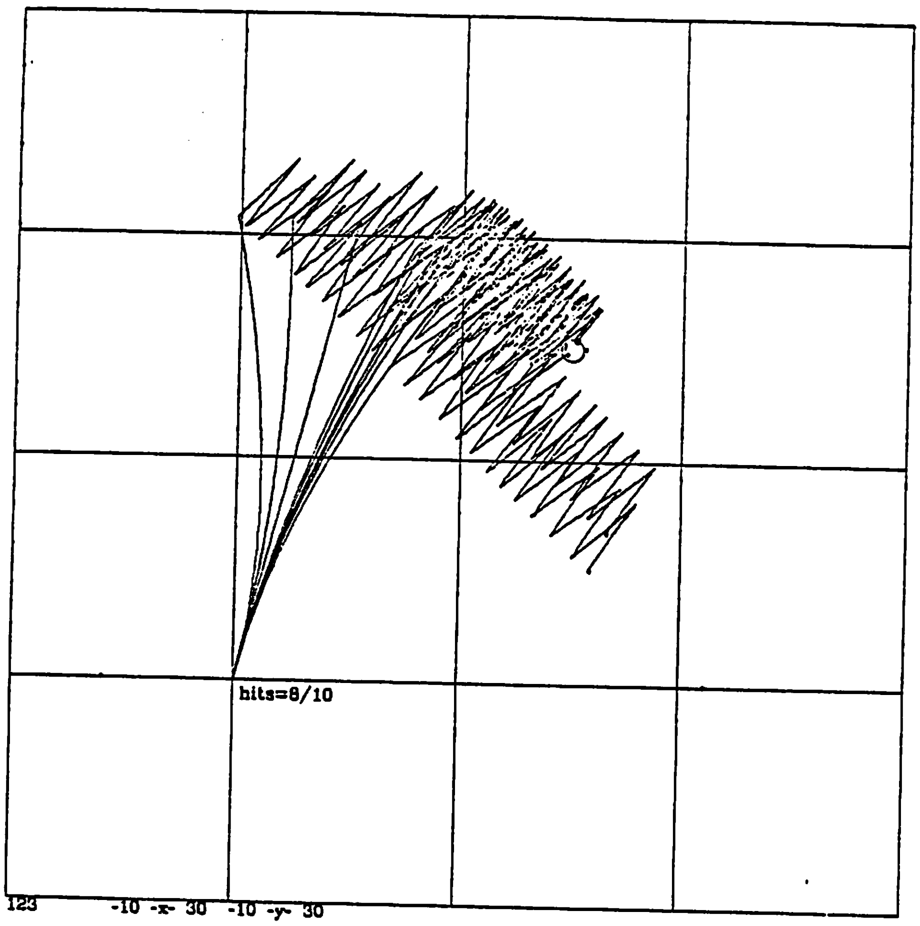

Figure Q13
Bad Angle Errors
Side Algorithm
Dumb Compass 2

fossum:
vplot

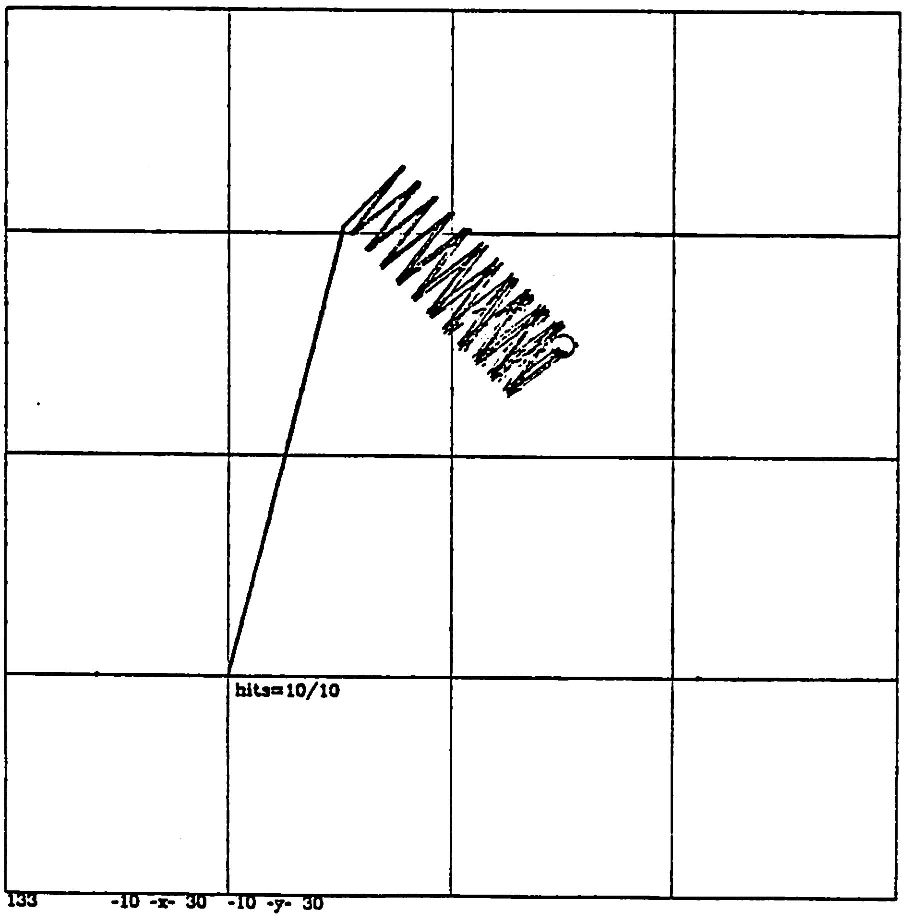

Figure Q14
Bad Angle Errors
Side Algorithm
Smart Compass

fossum:
vplot

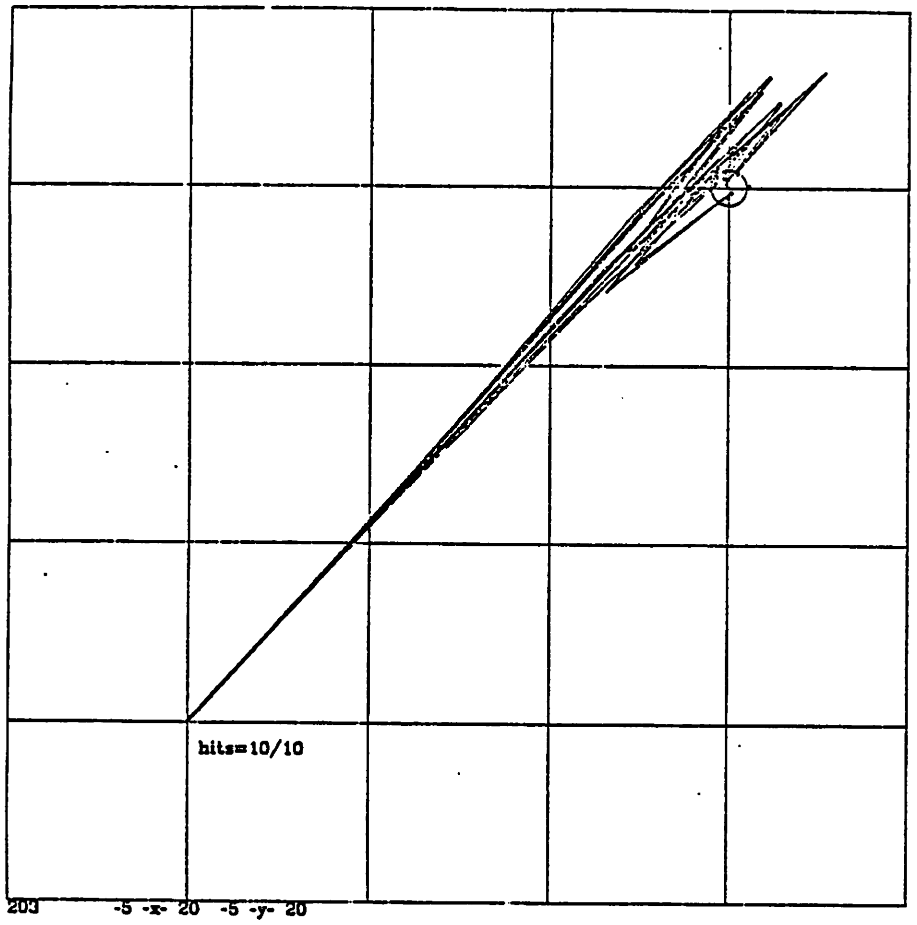

Figure Q15
Bad Path Errors
Side Algorithm
No Compass

fossum:
vplot

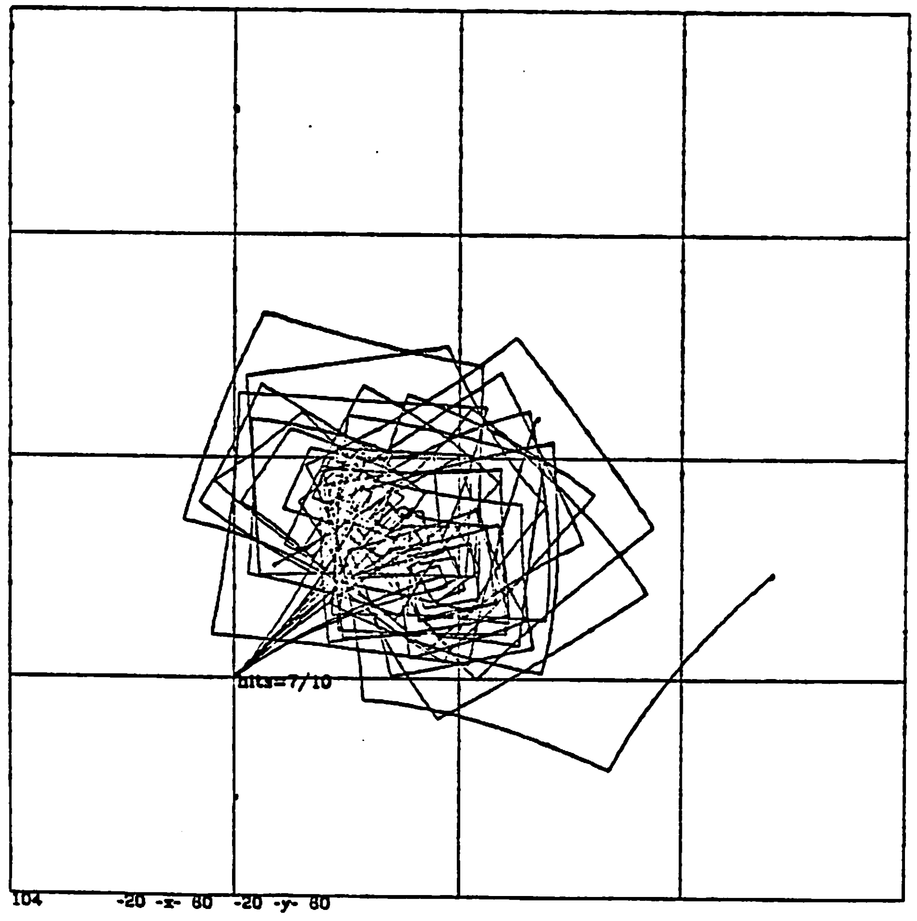

Figure Q16
Bad Angle Errors
Spiral Algorithm
No Compass

fossum:
vplot

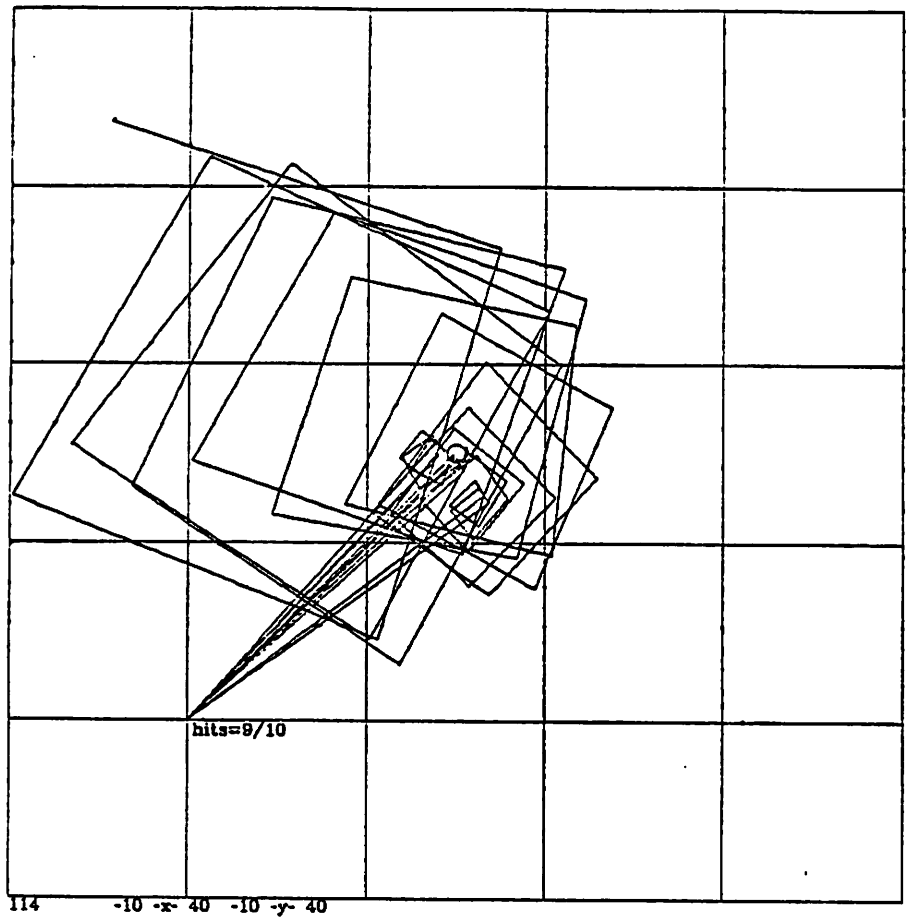

Figure Q17
Bad Angle Errors
Spiral Algorithm
Dumb Compass 1

fossum:
vplot

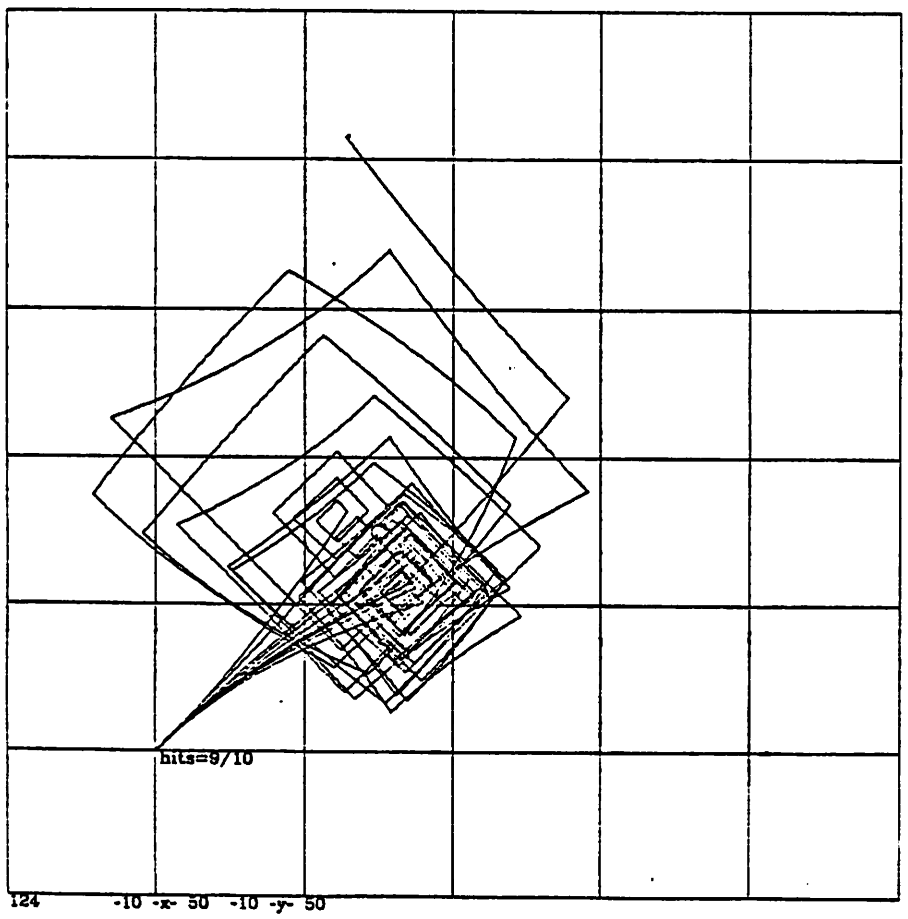

Figure Q18
Bad Angle Errors
Spiral Algorithm
Dumb Compass 2

fossum:
vplot

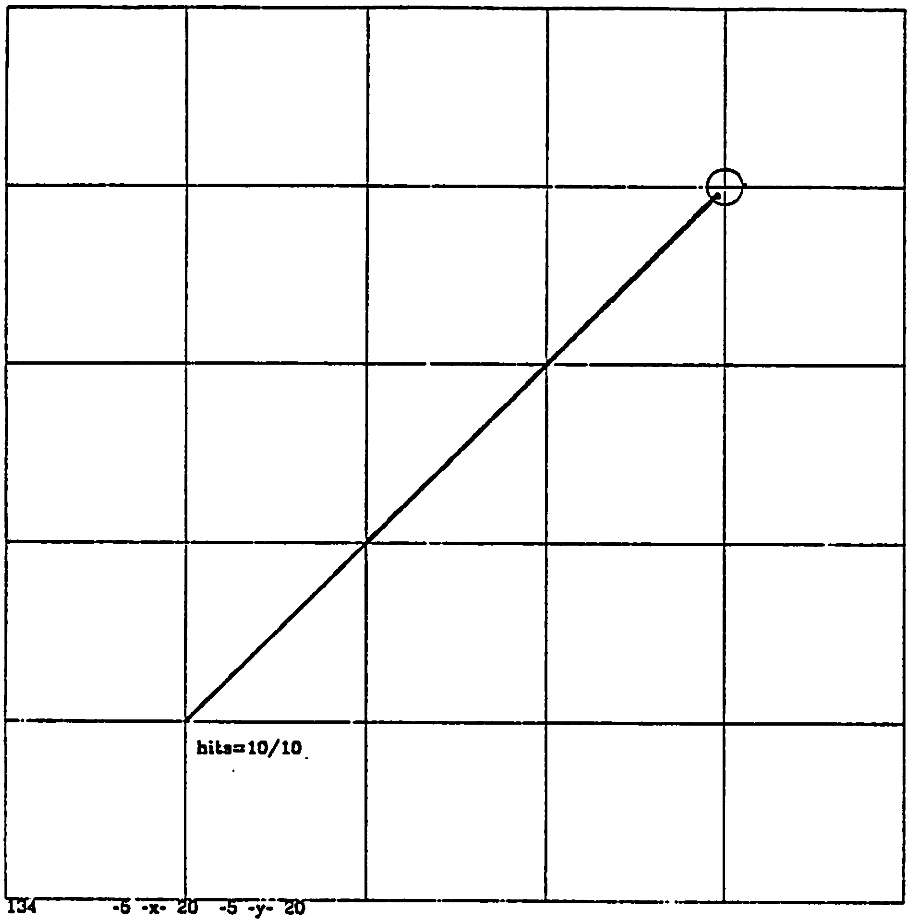

Figure Q19
Bad Angle Errors
Spiral Algorithm
Smart Compass

fossum:
vplot

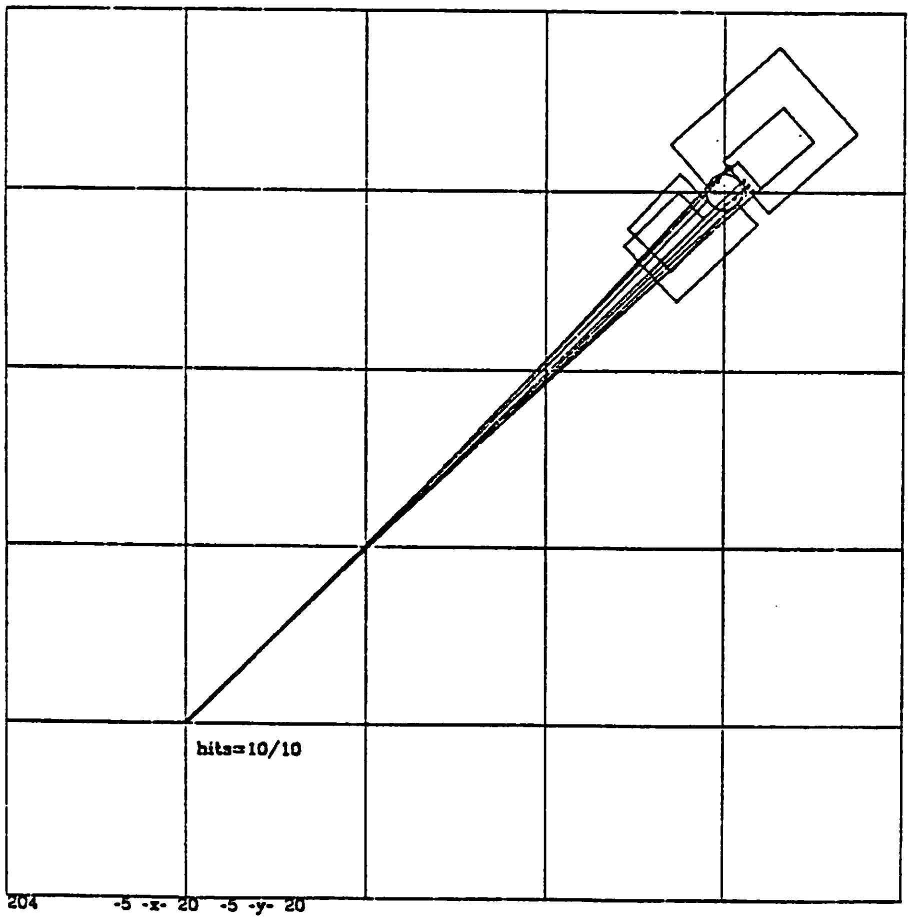

Figure Q20
Bad Path Errors
Spiral Algorithm
No Compass

fossum:
vplot

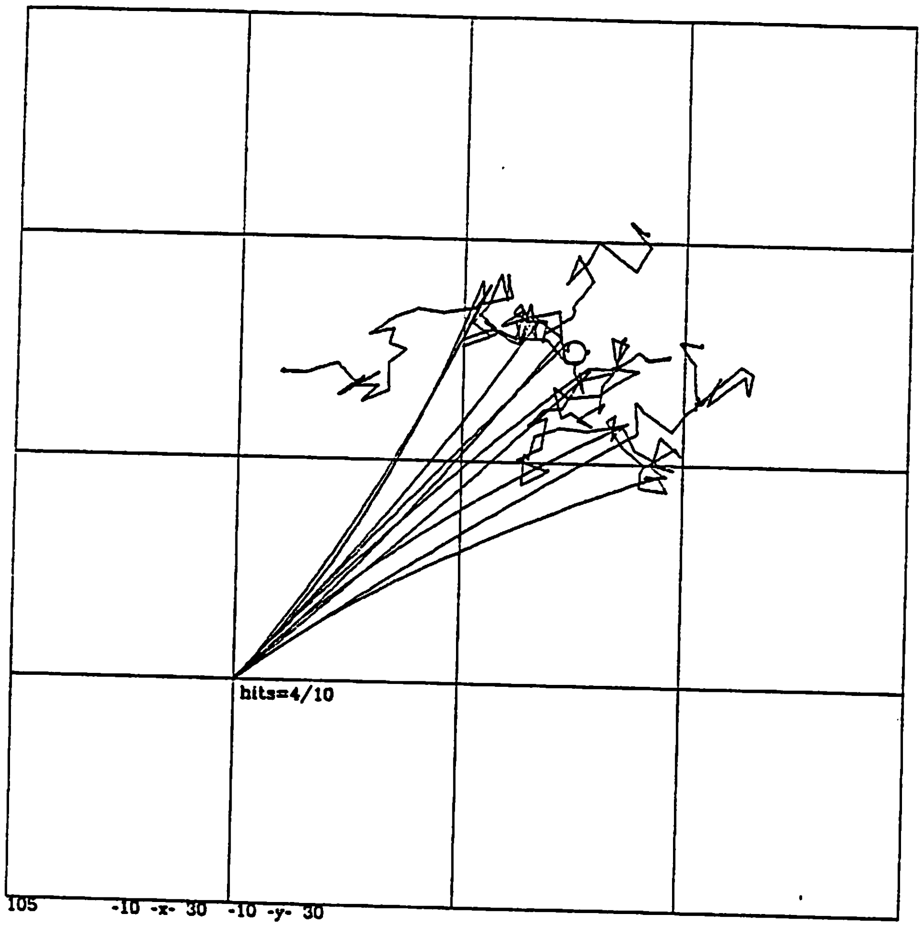

Figure Q21
Bad Angle Errors
Short Random Alg.
No Compass

fossum:
vplot

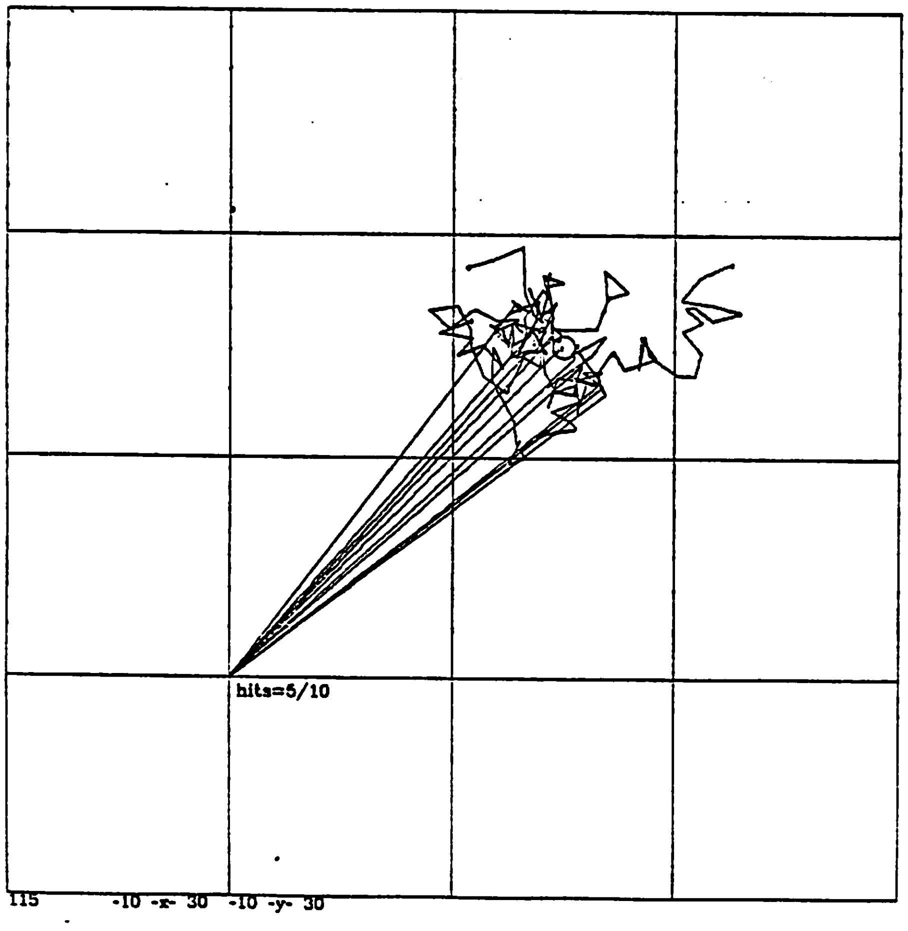

Figure Q22
Bad Angle Errors
Short Random Alg.
Dumb Compass 1

fossum:
vplot

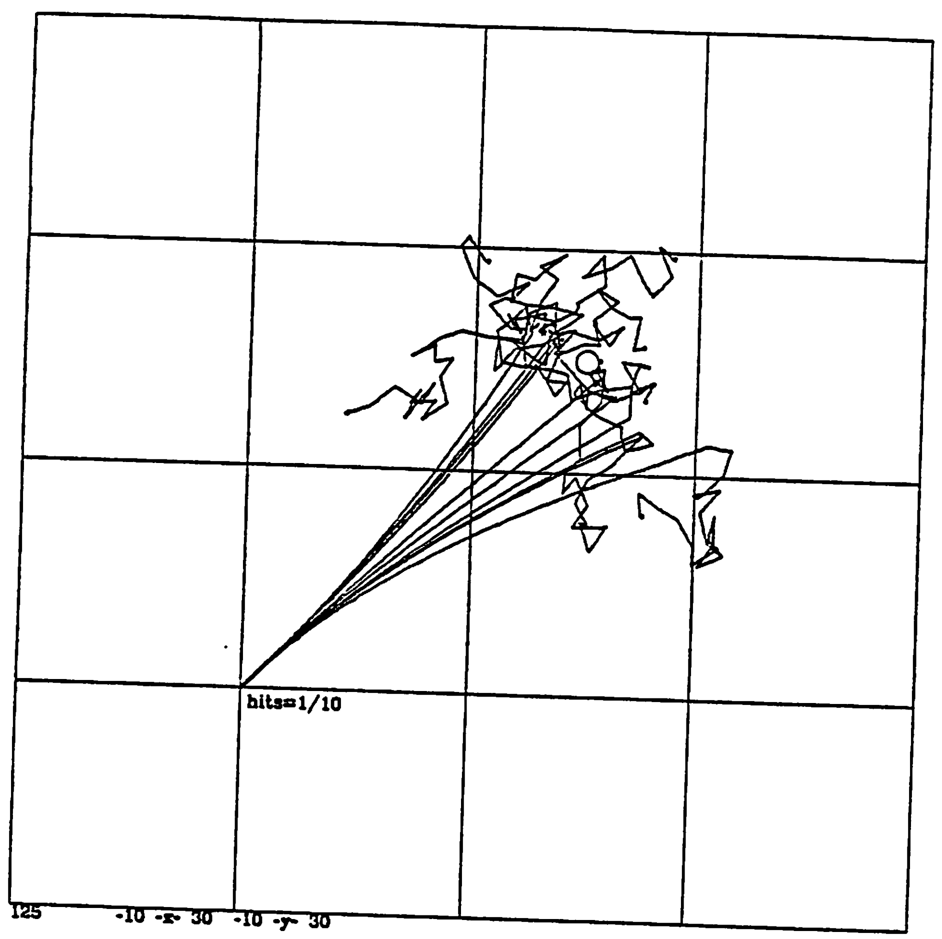

Figure Q23
Bad Angle Errors
Short Random Alg.
Dumb Compass 2

fossum:
vplot

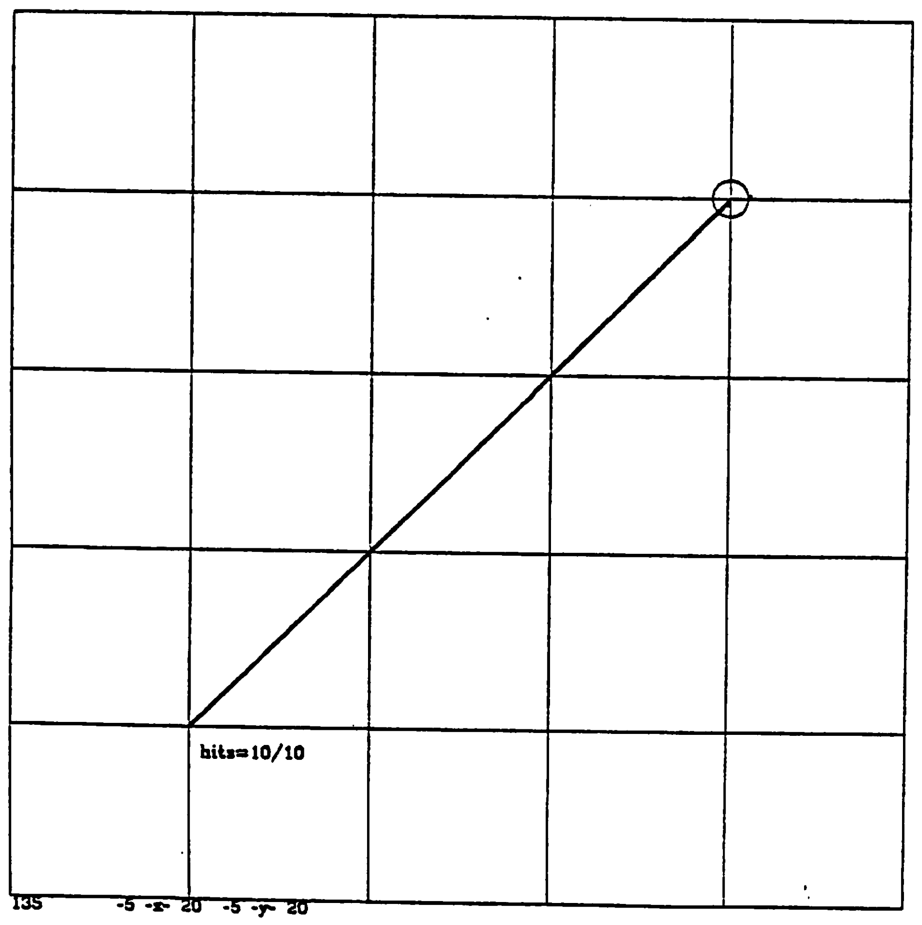

Figure Q24
Bad Angle Errors
Short Random Alg.
Smart Compass

fossum:
vplot

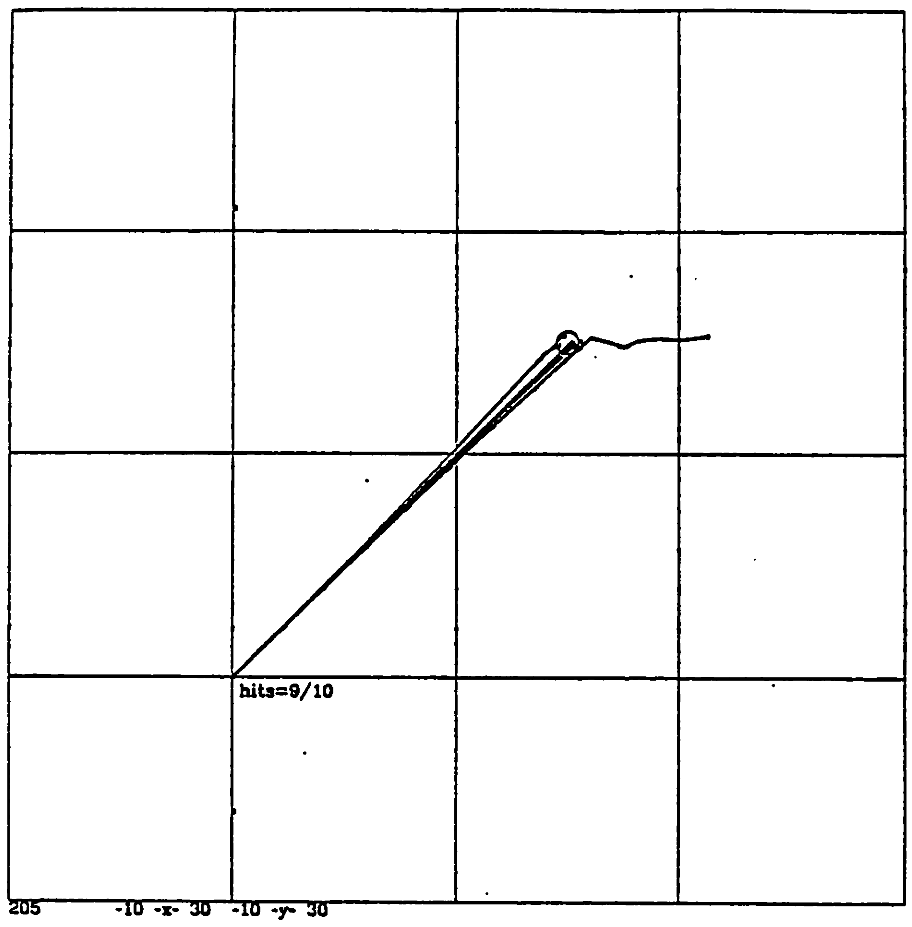

Figure Q25
Bad Path Errors
Short Random Alg.
No Compass

fossum:
vplot

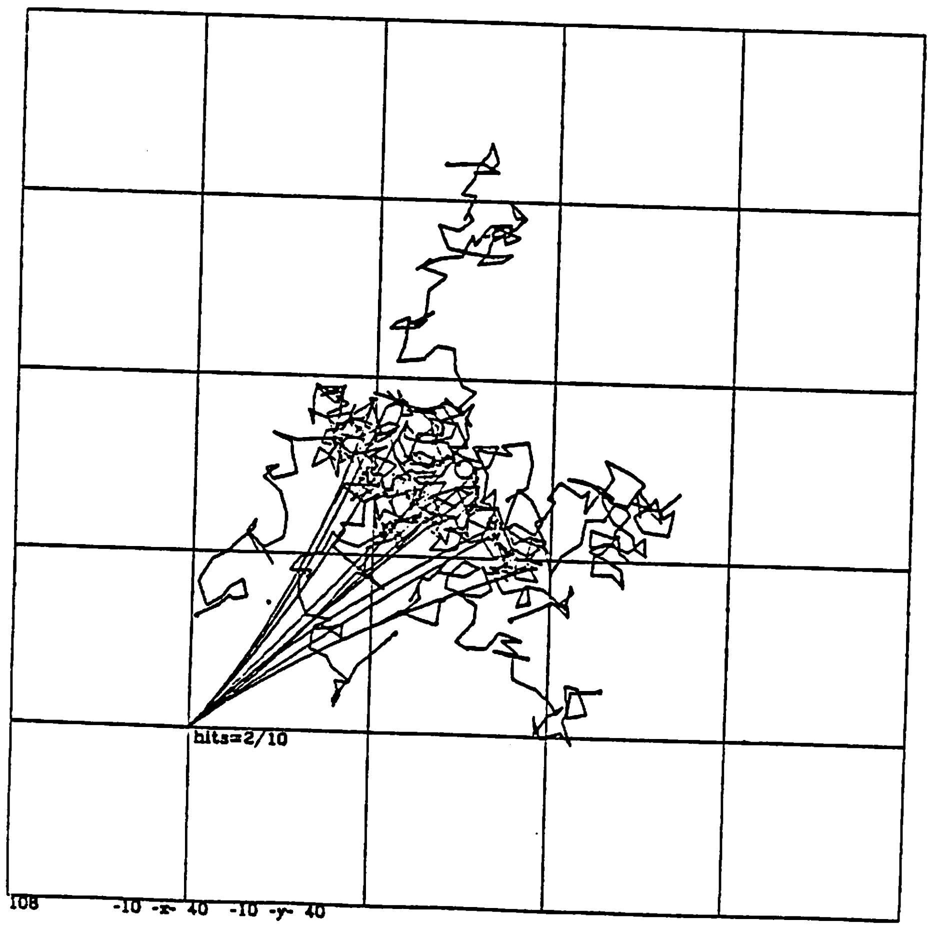

Figure Q26
Bad Angle Errors
Long Random Alg.
No Compass

fossum:
vplot

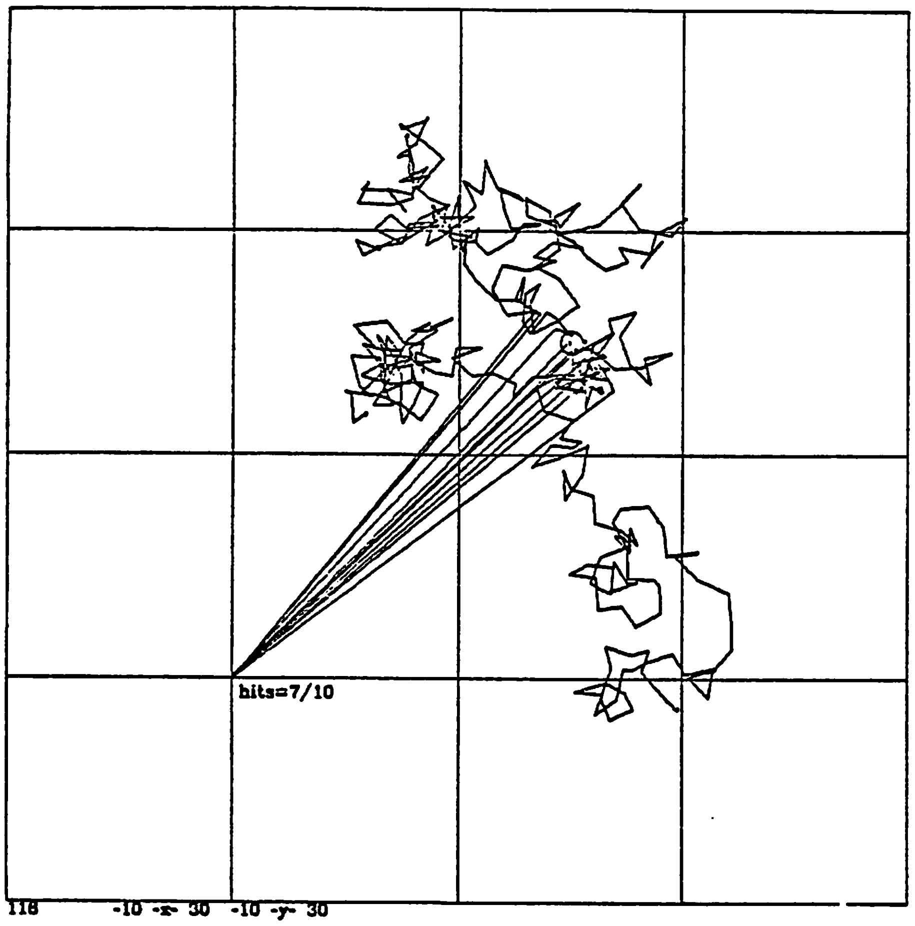

Figure Q27
Bad Angle Errors
Long Random Alg.
Dumb Compass 1

fossum:
vplot

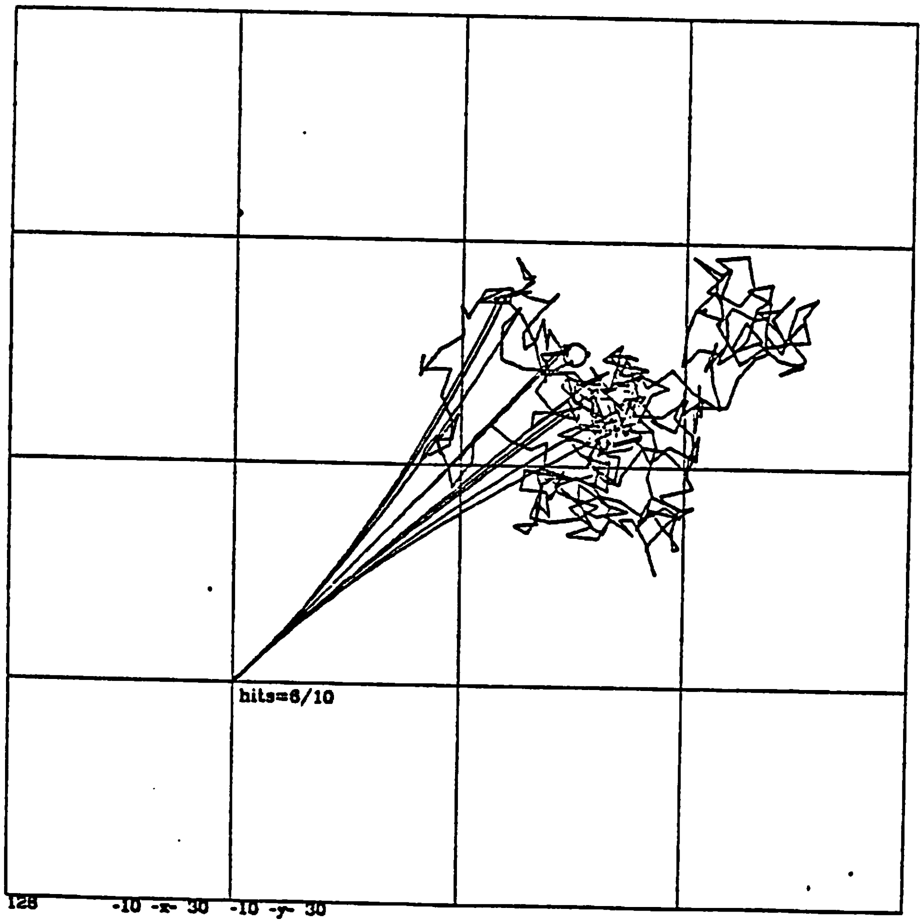

Figure Q28
Bad Angle Errors
Long Random Alg.
Dumb Compass 2

fossum:
vplot

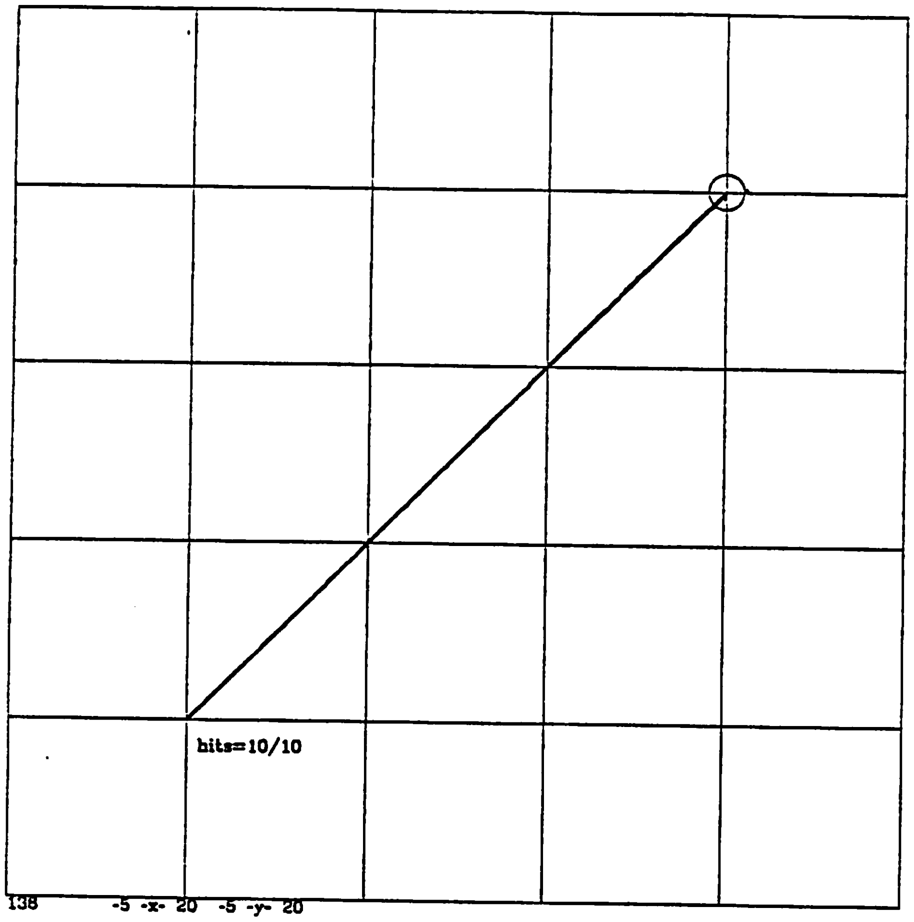

Figure Q29
Bad Angle Errors
Long Random Alg.
Smart Compass

fossum:
vplot

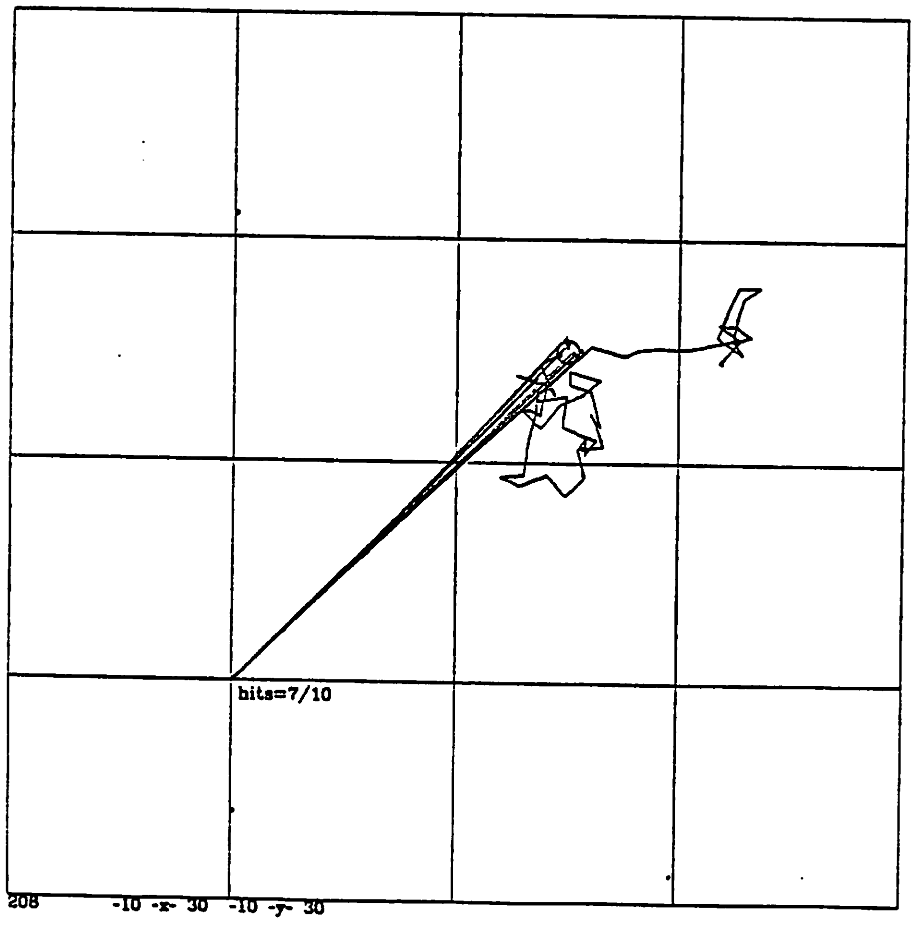

Figure Q30
Bad Path Errors
Long Random Alg.
No Compass

ERROR BOUNDS USED IN SIMULATION

	BAD ANGLE SCENARIO	BAD PATH SCENARIO
ABSSKID (meters)	0.01	0.01
RELSKID (m/deg)	0.0001	0.0001
ABSANGL (degrees)	5.0	0.05
RELANGL (unitless)	0.05	0.0005
ABSPATH (meters)	0.003	0.3
RELPATH (unitless)	0.001	0.1
MINCURV (meters)	30.0	300.0

FIGURE R

PERCENTAGE OF HITS

BAD ANGLE ERRORS

	No Compass	Dumb Comp 1	Dumb Comp 2	Smart Comp
Long	60.34%	71.06%	63.71%	100.00%
Short	49.18%	52.12%	40.76%	100.00%
Side	31.55%	32.63%	64.87%	100.00%
Spiral	81.45%	92.04%	88.03%	100.00%
S. Random	26.62%	55.65%	24.57%	100.00%
L. Random	35.71%	60.52%	36.36%	100.00%

BAD PATH LENGTH ERRORS

	No Compass	Dumb Comp 1	Dumb Comp 2	Smart Comp
Long	98.81%	100.00%	97.96%	100.00%
Short	99.37%	100.00%	99.85%	99.81%
Side	99.97%	99.42%	99.45%	98.46%
Spiral	94.13%	97.73%	92.91%	97.80%
S. Random	64.50%	74.14%	61.85%	71.28%
L. Random	69.58%	76.70%	68.61%	76.58%

Figure S

```
program target(input,output);                                          target

const pi=3.141592654;
        n=650;            {requested number of iterations}
        radius=0.5;       {radius of (circular) robot}
        smoothwt=50.0;    {weight factor for non-linearity calculations}
        comperr=0.1;      {error of compass, in degrees}
        comptol=0.3;      {tolerance of compass, in degrees}

type insptr=^instruction;
      instruction=record          {each of these records }
                    rotn,tran:real;  {will hold one command }
                    next:insptr      {a rotation part, and a}
                    end;              {translation part.      }

var instrptr:insptr;   {points to record containing present command}
     head:insptr;        {points to record containing first command}
     hit:boolean;        {becomes true whenever target is hit}
     complvl:integer;    {0=nocomp; 1=dumbcomp1; 2=dumbcomp2; 3=smartcomp}
     count:integer;      {counts number of hits}
     i,:integer;         {used as an index in for loops}
     xnow:real;          {x-coordinate of robot's present position}
     ynow:real;          {y-coordinate of robot's present position}
     headingnow:real;    {robot's present heading, in degrees}
     rotncmd:real;       {rotation command, as read from linked list}
     trancmd:real;       {translation command, as read from linked list}
     storedhdg:real;     {what the heading should be, according to compass}
     mincurv:real;       {radius of smallest circle robot could travel}
     xtarget:real;       {x-coordinate of target}
     ytarget:real;       {y-coordinate of target}
     absskid:real;       {maximum value of absolute skid}
     relskid:real;       {maximum value of relative skid}
     absangl:real;       {maximum value of absolute angular error}
     relangl:real;       {maximum value of relative angular error}
     abspath:real;       {maximum value of absolute path length error}
     relpath:real;       {maximum value of relative path length error}

procedure buildlist;        {this procedure is called only when all}    buildlist
                                {that's left in the input file is a list}
var r,t:real;               {of commands; these are read into a}
     ptr:insptr;            {linked list by this procedure.}

begin
if eof(input)
then writeln 'no instructions.')
else begin
       new(instrptr);
       ptr:=instrptr;
       repeat
              readln(r,t);
              new(ptr^.next);
              ptr:=ptr^.next;
              ptr^.rotn:=r;
              ptr^.tran:=t
              until eof(input);
       ptr^.next:=nil;
       instrptr:=instrptr^.next
       end
end;

procedure move(mag,dir:real);                                          move
```

```
var xnew:real;    {x-coordinate of robot's location after move}
    ynew:real;    {y-coordinate of robot's location after move}
    temp1:real;   {}
    temp2:real;    {these are temporary variables}
    parm:real;    {}

begin
xnew:=xnow+(mag *cos(pi *dir/180.0));
ynew:=ynow+(mag *sin(pi *dir/180.0));
if mag<>0.0
then begin
      if sqr(xnew-xtarget)+sqr(ynew-ytarget)<sqr(radius)
      then hit:=true

      {the two lines above check to see if the robot has hit the target}
      {at the END of the move.  The twelve lines below are required to }
      {check whether the robot hit (or, rather, grazed) the target}
      {DURING the move.}

      else begin
            temp1:=(ytarget-ynow) *(ynew-ynow);
            temp2:=(xtarget-xnow) *(xnew-xnow);
            parm:=(temp1+temp2)/sqr(mag);
            if (0<parm) and (parm<1)
            then begin
                  temp1:=(1-parm) *xnow+parm *xnew;
                  temp2:=(1-parm) *ynow+parm *ynew;
                  if sqr(xtarget-temp1)+sqr(ytarget-temp2)<sqr(radius)
                  then hit:=true
                  end
            end;
      xnow:=xnew;
      ynow:=ynew;
      if n<21 then writeln(xnow,ynow);
      if hit=true then count:=count+1
      end
end;
```

```
procedure xeqcmd;

var mag:real;               {contains magnitude of "skid" moves}
    dir:real;               {contains direction of "skid" moves}
    incrlength:real;        {contains length of incremental moves}
    distance:real;          {contains path length, with errors}
    deltahead:real;         {contains incremental change in heading}
    maxdeltahead:real;      {contains maximum possible 'deltahead'}
    incr:integer;           {counts the number of incremental moves so far}

begin
rotncmd:=instrptr^.rotn;
trancmd:=instrptr^.tran;
instrptr:=instrptr^.next;
if rotncmd<>0.0
then begin

      {here, the robot has been told to rotate, so we calculate and}
      {incorporate some errors tied to the rotation.  These are the}
      {absolute skid, relative skid, absolute angle error, relative}
      {angle error, and compass error}

      if complvl>1 then storedhdg:=storedhdg+rotncmd;
      dir:=360.0 *random(0.0);
      mag:=absskid *random(0.0);
      move(mag,dir);
```

```
          dir:=360.0*random(0.0);
          mag:=relskid*rotncmd*random(0.0);
          if not(hit) then move(mag,dir);
          headingnow:=headingnow+rotncmd*(1+relangl*(random(0.0)*2.0-1.0));
          headingnow:=headingnow+absangl*(random(0.0)*2.0-1.0);
          mag:=(2.0*random(0.0)-1.0)*comperr;
          if complvl=1 then storedhdg:=headingnow+mag;
          if complvl>1 then headingnow:=storedhdg+mag
          end;
     if trancmd>0
     then begin

          {here, we calculate the (erroneous) path length, compute the length}
          {of the incremental moves to be made, and the worst incremental     }
          {changes in heading possible, given 'mincurv' and 'incrlength'.     }

          distance:=trancmd*(1+relpath*(random(0.0)*2.0-1.0));
          distance:=distance+abspath*(random(0.0)*2.0-1.0);
          incrlength:=sqrt(mincurv*trancmd)/60.0;
          incr:=1;
          maxdeltahead:=(180.0*incrlength)/(pi*mincurv);
          deltahead:=maxdeltahead*(random(0.0)*2.0-1.0);
          while (incr*incrlength<distance) and not(hit) do
               begin

               {in this inner loop we accomplish an incremental move, adjust}
               {our heading slightly, and repeat until either the target is }
               {hit, or we've traversed the entire path length of the command.}

               move(incrlength,headingnow);
               headingnow:=headingnow+deltahead;
               if (abs(headingnow-storedhdg)>comptol) and ((complvl=1)or(complvl=3))
               then headingnow:=storedhdg+(2.0*random(0.0)-1.0)*comperr;
               deltahead:=smoothwt*deltahead+maxdeltahead*(random(0.0)*2.0-1.0);
               deltahead:=deltahead/(smoothwt+1.0);
               incr:=incr+1
               end;
          if not(hit)
          then begin  {this takes care of the last little fractional piece.}
               incrlength:=distance-(incr-1)*incrlength;
               move(incrlength,headingnow)
               end
          end
     end;

     begin
     readln(complvl);
     readln(absskid);
     readln(relskid);
     readln(absangl);
     readln(relangl);
     readln(abspath);
     readln(relpath);
     readln(mincurv);
     readln(xtarget);
     readln(ytarget);
     count:=0;
     buildlist;
     head:=instrptr;
     if n<21 then for i:=1 to 120 do {this draws the target}
     writeln(xtarget+radius*(cos(i*pi/60)),ytarget+radius*(sin(i*pi/60)));
     writeln(xtarget+radius*(cos(pi/60)),ytarget+radius*(sin(pi/60)),'  ,');
     for i:=1 to n do
          begin
```

```
        if sqr(xtarget)+sqr(ytarget)<sqr(radius)
        then begin
             hit:=true;
             count:=count+1
             end
        else hit:=false;
        xnow:=0.0;
        ynow:=0.0;
        headingnow:=90.0;
        if complvl>0 then storedhdg:=90.0;
        instrptr:=head;
        while not(hit) and (instrptr<>nil) do xeqcmd; {THE loop!}
        if n<21 then writeln(xnow,ynow,' .');
        end;
writeln(0.3,-0.9,' hit percentage=',float(count)/float(n):4:2,'%')
end.
```