new computer communication switching technique. *Computer Network*, **3**, 267–268 (1979).
8. Occam User Group Newsletter, no. 10 (Jan 1989).
9. Vijay Ahuja, *Design and Analysis of Computer Communication Network*. McGraw-Hill Book Company (1982).
10. A. E. Knowles, *Specification for T-Rack* PSF/MU/87/AEK/3 Internal report, Department of Computer Science, University of Manchester (1986).
11. INMOS, *Transputer Technical Reference* Prentice Hall (1989).
12. N. T. Son, An Approach to the dynamic Reconfiguration of a Multi-transputer Network. PSF/PCL/WP6/89/3 Parallel Computing Research group, Polytechnic of Central London.
13. Nicos Christofides, *An Algorithmic Approach*. Academic Press, London, New York, San Francisco.
14. M. Bozygit, A dense variable topology multicomputer system. Ph.D. Thesis submitted at Polytechnic of Central London.
15. Dharma P. Agrawal *et al.* Evaluating the performance of Multicomputer Configurations. *Computer*, **19** (5), 23–37 (1986).
16. Clifford Wallace Marshall, *Applied Graph Theory*. Wiley Interscience (1971).

Correspondence

Does a Point Belong to a Polygon?

Sir,

To describe a space region when solving a boundary problem I wrote a program determining whether a point belongs to a polygon specified by the coordinates of its own vertices. My algorithm when compared with similar ones, in particular with that by R. Francline,[1] turns out to be faster. To my knowledge, such an algorithm has not yet been published. Its essence lies in the following.

In the given polygon, one vertex is singled out, and the remaining ones are connected to it. Any two adjacent vertices and the one singled out make a triangle. The point may belong to one or several triangles or to none of them. If the number of triangles to which the given point belongs is even, then the point lies outside the polygon, otherwise it lies inside it. The algorithm seems at first sight to be more cumbersome than the traditional one, based on counting the number of the points crossing the polygon boundary by a ray drawn from the point in any direction, and not well suited for writing an efficient program. However, the comparison shows that this is not the case.

If a polygon has N vertices and, consequently, N sides, then, according to the traditional algorithm, one needs N tests to find out whether the chosen ray crosses the line, a part of which belongs to the polygon boundary. On the average, the number of these crossings should be $N/2$; moreover, one should check whether the crossing point belongs to the boundary, i.e. whether it lies between the vertices. Then the average number of basic checks will be $\frac{3}{2}N$.

In the algorithm proposed, $N-1$ checks are made to find out on which side of the rays, emerging from the given vertex and passing through the other vertices of the polygon the point lies. If the point lies between two rays, one checks whether it is inside the triangle. The number n of such checks is seldom more than 3 and, on the average, is less than 1. (The figure presents the rare difficult case.) The number of basic checks is about N.

The necessity in additional checks for 'special cases', when the point is on the boundary or on the line belonging to it, or in the polygon vertex, increases the ratio of the number of checks. In practice, the program presented in the Appendix runs almost twice as fast as the one by R. Francline.[1]

I mention the testing technique since it is right at the point where we encounter discrepancies when estimating these algorithms. The running time of the complete test program consists of the running time τ of the program to be tested plus the running time t of the remainder of the test program. Therefore if the test program runs 2.1 times faster with the program proposed here included, as compared with the case when Francline's program is included, this does not necessarily mean that their efficiencies differ by more than two times. Here we have

$$(t+\tau_F)/(t+\tau_K) \sim 2.1.$$

If in the same test program we have two calls to the program to be tested rather than one, it is quite possible that one can obtain

$$(t+2\tau_F)/(t+2\tau_K) \sim 1.9,$$

from which one can calculate $\tau_F/\tau_K \sim 1.75$. These are the numbers obtained by me from numerous tests of the programs under discussion.

The sequence of the formulae composing this algorithm can easily be understood from the program given in the Appendix.

The general algorithm, applied to domains of arbitrary dimensions bounded with arbitrary surfaces (in the two-dimensional case we have lines), is described in Ref. 2, and one of its modifications will probably be published in the Soviet journal *Programming*.

The general algorithm has already proved its efficiency, for example in determining the location of the detector matter penetrated by a particle. This is very important when dealing with the huge number of particles handled by experimental high-energy physics.

Yours faithfully,

P. A. KALINCHENKO
USSR 142284, Moscow region, Protvino,
Institute for High Energy Physics,
Computer Centre

References

1. M. Smith, Points, polygons, and areas (letter to the Editor). *The Computer Journal*, **23** (2), 189 (1980).
2. P. A. Kalinchenko, Preprint, Institute for High Energy Physics **86-60**. Serpukhov (1986).

Appendix

```
      SUBROUTINE PPOLYN
     *  (X, Y, XP, YP, N, IV)
C     30 OCTOBER 1985
C     WRITTEN BY KALINCHENKO
      DIMENSION XP (N), YP (N)
      N1=N-1
      XPN=XP(N)
      YPN=YP(N)
      XN=X-XPN
      YN=Y-YPN
      IV=1
      P=1.
      ISTART=-1
      I=0
    1 IQ=-1
    2 I=I+1
      IF (I-N1)  4, 3, 16
    3 ISTART=-N
    4 XIN=XP(I)-XPN
      YIN=YP(I)-YPN
      IF ((XIN*YN-YIN*XN)*P)
     *  5, 7, 2
    5 P=-P
      IF (ISTART+1)  9, 2, 9
    7 IF (ISTART+I)  11, 11, 8
    8 IQ=1
    9 XIJ=XP(I)-XP(I-1)
      YIJ=YP(I))-YP(I-1)
      XJ=X-XP(I-1)
      YJ=Y-YP(I-1)
      IF ((XIJ*YJ-YIJ*XJ)*P)
     *  10, 12, 1
   10 IV=IQ*IV
      GO TO 1
   11 XIJ=XIN
      XJ=XN
      YJ=YN
   12 IF (XIJ)  13, 14, 13
   13 XI=X-XP(I)
      IF (XI*XJ)  (15, 15, 1
   14 YI=Y-YP(I)
      IF (YI*YJ)  15, 15, 1
   15 IV=0
   16 RETURN
C        -1-INSIDE
C    IV: 0-EDGE OR VERTEX
C          1-OUTSIDE
      END
```

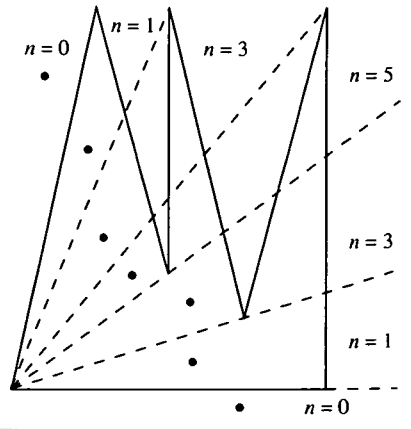

Fig. 1.