Program Promenade

連載

多角形の面積の近似

石畑　清（明治大学理工学部）
ishihata@cs.meiji.ac.jp

今回も幾何の問題を取り上げる．筆者の担当分では3回目ということになる．

幾何の問題としては，これまでに取り上げたものに比べて，やさしいといってよいと思う．難しい数式を持ち出す必要はないし，切れ味鋭い幾何的な考察を要するわけでもない．数式としては，せいぜい直線の方程式（1次式）が出てくるだけである．プログラミングの面から見ても，難しい問題の部類には属さない．コードの量はさほど多くならないし，アルゴリズムが難しいわけでもない．

しかし，それならやさしいかと言われると，そうも言い切れない．プログラミングに際して注意しなければならない点があるからである．うっかりすると陥りそうな罠と言い換えてもよい．いかにしてそれを避けるかを中心に説明していこうと思う．

■問題

今回取り上げる問題は，2003年11月に行われた会津大会の問題C「Area of Polygons」である．平面上に与えられた多角形の面積の近似値を求めることが課題である．

図-1 のように，座標平面上に多角形が与えられている．多角形の頂点の座標はすべて整数である．この多角形の面積を概算でよいから求めたい．その方法としては，正方形の個数を勘定することで，およその面積の数値を得る手法を採用する．多角形に少しでも重なる正方形の個数を数えるのである．

図-1に示すように，多角形を含む平面上に格子線を引く．すなわち，x軸とy軸に平行に，それぞれ座標値が整数の直線を何本も引く．すると，小さな正方形がたくさんできるが，このうちで多角形に少しでも重なる正方形の個数を数える．図では，このような正方形を網掛けで表現している．図-1の例の場合は，網掛けの正方形が55個あるので，55が答になる．このような近似面積を求めることがこの問題の要求である．

多角形に少しでも重なる正方形とは，多角形に完全に含まれるものか，多角形の辺が通っているものかのいずれかである．問題文では，多角形との共通部分の面積が0でないものと表現している．こう表現すれば間違いの余地はなかろう．多角形と頂点を共有するだけの正方形は勘定に入らない．

縦横の線だけから構成されている場合を除けば，こうして求めた近似面積は，真の面積より大きい．また，近似として精度がよいともいえない．さらに，プログラムを使って面積を求めることを考えるのであれば，多角形の本当の面積の方が上述のような近似面積よりはるかにやさしい．幾何の問題を解くプログラムを少しでも勉強したことのある人なら，多角形の真の面積を求めるプログラムは，ほんの演習問題程度のものでしかない．

つまり，この問題に実用的な価値はほとんどない．問題文には，数学の勉強の手助けをするというストーリーが書いてあるが，これも怪しげである．プログラミングの問題と割り切って，解を得るまでの思考を楽しむことにしよう．

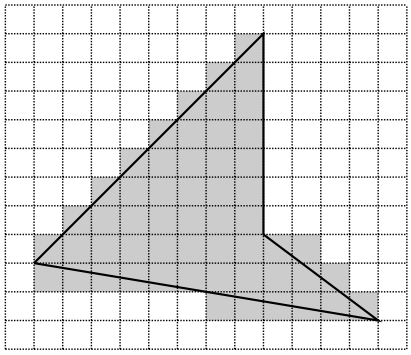

図-1　入力データと解答の例

プログラムの入力は，次のような形式になる．

n
$x_1\ y_1$
$x_2\ y_2$
$\cdots$
$x_n\ y_n$

最初の行が頂点の個数 n である．n は 100 以下と定められている．2 行目以降に多角形の頂点の座標を順に並べてある．それぞれの k に対して，k 番目の頂点と k+1 番目の頂点が辺で結ばれている（$1 \leqq k \leqq n$）．また，最後の n 番目の頂点と 1 番目の頂点も辺で結ばれている．座標値は整数に限られていて，x，y とも −2000 以上，+2000 以下である．

多角形は，凸多角形の場合もあるし，図 -1 のような凹多角形の場合もある．辺と辺が頂点以外の場所で交差したり，同じ位置に複数個の頂点があったりすることはない．多角形としてはごく素直なものだけを考えればよい．

頂点の座標を与える順番は，時計回りのこともあるし，反時計回りのこともある．どちらかに決まっていれば，プログラミングが少し楽になる可能性もあるのだが，出題者はそこまで親切にはしてくれなかった．

以下のプログラムでは，入力データを次の変数に読み込んであるものと仮定する．

```
int n;
struct { int x, y; } a[Max_Vertex+1];
```

それぞれの変数およびメンバの意味は明らかだろう．Max_Vertex は 100（頂点数の最大）を表す #define 定数である．配列の長さを 1 つ多くとっているのは，a[0] を a[n] にコピーしてから計算を始めるからである．こうすることによって，n 番目の点と最初の点を結ぶ辺を特別扱いせずに済んで，プログラムが簡単になる．

■ Java の AWT を利用した解法

最初に，ほとんどプログラミングをせずに済ます方法を紹介しよう．いわば，ずるい解法である．審判としては，こんなプログラムを書かれては困るのだが，いかにも簡単なプログラムなので，紹介しないわけにはいかない．

Java には AWT（abstract window toolkit）という名のパッケージがある．このパッケージは，ユーザインタフェースを作ったり，図形やグラフィクスを扱ったりするときに便利な機能を豊富に備えている．その中には，図形と図形の重なり具合を調べるための関数も含まれていて，これを使えば，今回の問題などは簡単に解ける．

実際に利用できるのは，GeneralPath という名のクラスである．これは，何本かの直線または曲線が連なってできる経路を表し，それに関連するいくつかの操作を提供している．このクラスを使うためには，プログラムの先頭に次のような import 宣言を置く必要がある．

```
import java.awt.geom.GeneralPath;
```

このクラスは，次のようなメソッドを提供している．ただし，path はこのクラスの変数である。

path.contains(x, y, w, h)
: 長方形が path で囲まれる図形に含まれるか否かを調べる．長方形の x 座標は x 〜 x+w（w が幅），y 座標は y 〜 y+h（h が高さ）である．4 つの引数の型は，いずれも double である．

path.intersects(x, y, w, h)
: 長方形が path で囲まれる図形と少しでも重なるか否かを調べる．

初めに，与えられた頂点の座標から，変数 path が多角形を表すように初期化する．その方法は次のとおりである．

```
path = new GeneralPath();
path.moveTo(a[0].x, a[0].y);
for(i = 1; i < n; i++)
  path.lineTo(a[i].x, a[i].y);
path.closePath();
```

メソッド moveTo と lineTo を使って多角形の頂点を順に結んでいき，最後の closePath で閉じた曲線にしている．

ここまでできれば，後は簡単である．たとえば，次のようなプログラムでも原理的には答が求まる．

```
area = 0;
for (x = -2000; x < 2000; x++)
  for (y = -2000; y < 2000; y++)
    if (path.intersects
          ((double)x, (double)y, 1.0, 1.0))
      area++;
```

問題で要求しているとおりに，重なりを持つ小正方形の個数を数えているだけである．本来なら，正方形が多角形に含まれるかどうか，線と交差するかどうかなどを調べる部分のプログラミングが厄介なのだが，このプログラムでは Java のライブラリ任せにして，何も考えずに済ませてしまったことになる．これでよければ，こんな簡単なことはない．

上のプログラムでは，メソッド intersects を1600万回呼ぶことになる．このメソッドは，一般の図形を扱えるように作られているので，かなり時間がかかる．1600万回も呼んだのでは時間オーバーになる恐れが大きい．

これを避けるために，x と y それぞれについて，最小値と最大値を求め，この2つの値の間だけ調べるようにすることが考えられる．ループの回数が減って，それだけ速くなるであろう．しかし，目一杯大きな図形がたくさん与えられれば役に立たない．

より本質的な解決策は，分割統治法を利用することである．この連載でも何度か取り上げた（たとえば2002年9月号）が，幾何の問題を解く方法として，分割統治法が有効である場合が多い．図形全体がある性質を持つかどうか判断できない場合，それを縦横それぞれ半分に切って4つの図形に分ける（平面図形の場合）．そして，その4つの部分それぞれについて性質を調べる．この手順を図形それぞれが十分に小さくなるまで繰り返す方法である．

この問題の場合，分割統治法によって近似面積を求める関数は次のようになる．

```
int computeArea(int x, int y, int size)
{
  if (!path.intersects
        ((double)x, (double)y,
         (double)size, (double)size))
    return (0);
  else if (size == 1)
    return (1);
  else if (path.contains
            ((double)x, (double)y,
             (double)size, (double)size))
    return (size*size);
  else {
    int hs = size/2;
    return (computeArea(x, y, hs)
          + computeArea(x, y+hs, hs)
          + computeArea(x+hs, y, hs)
          + computeArea(x+hs, y+hs, hs));
  }
}
```

大きな正方形が丸ごと多角形に含まれている場合は，それ以上細かく分けずに正方形の大きさをそのまま面積とする．また，正方形と多角形が重なりを持たない場合は面積0である．それ以外の場合（部分的に重なっている場合）は，縦横それぞれ半分に切って，4つの部分それぞれに同じ手続きを繰り返す．正方形の大きさが1になったら，部分的な重なりでも面積1として，それ以上分割しない．

この関数の最初の（main からの）呼出しは次のようにすればよい．

```
area = computeArea(-2048, -2048, 4096);
```

この方法なら，大きな図形でもそれほどの時間はかからない．

■ AWT 使用の是非

以上のように，Java の AWT を使えば，問題を簡単に解けることが分かった．しかし，こんな解法が許されるようではプログラミングコンテストが成り立たない．言語によるハンデが生じるからである．コンテストでは，プログラミング言語として，C，C++，Java，Pascal のどれを選んでもよいとしている．そのうちの Java だけに簡単な解法があって，他の言語にないのでは明らかに不公平である．

また，このような解法では，本当にプログラムを作ったことにならないという意見もあると思う．連載第1回（2002年4月）にあるように，最近は自分でプログラムを作らずに，既存のプログラムを再利用することが盛んである．再利用を促進する技術が進歩してきたからで，それはそれで正しい方向ではあるのだが，やはりプログラミングコンテストが求めている能力とは少し違うだろう．コンテストが再利用の技術を問う方向に変わっていく可能性がないとはいえない．しかし，少なくとも現時点でのコンテストが要求するのは自分でプログラムを書く能力である．

この問題が AWT で簡単に解けることは，審判にも分かっていた．それを禁止するような対策を講じなければならないのだが，その方法はいくつか考えられる．2002年金沢大会の問題 H「Viva Confetti」（2003年6月号の本連載参照）の場合は，問題文の中で AWT の利用を禁止した．これに対して，この問題の場合は，判定用データの工夫で対処した．分割統治法を利用した速いプログラムでも，時間オーバーになるような，大きくかつ複雑な図形を多数用意したのだ．分割統治法よりさらに上手な AWT 利用法があれば突破されるかもしれないが，そんなものがあるとは思えないので，実質的に AWT を禁止したことになる．

このような間接的な禁止の仕方が，審判として適切な態度かどうかは分からない．AWT では解けないことを知らずに挑戦するチームが現れて，無駄に時間を費やす羽目になる恐れがある．問題文で禁止する方が直接的で紛れがないことは確かだろう．しかし，問題文に AWT 禁止などと書き込みたくないという気分も理解できる．問題そのものからかけ離れた異質な記述だからである．

Java には，AWT のほかにも言語間のハンデの原因となり得るパッケージがある．たとえば，BigInteger（多倍長整数）である．もちろん，審判はこれによる不公

平も生じないように気を配っている．基本的な態度は，整数計算なら普通の 32 ビット符号付き整数で間に合うように作題することである．これなら，BigInteger があろうとなかろうと関係ない．昔と比べて，審判が考えなければいけない諸事情が増えて，なかなか厄介な時代になってきたといえよう．

正直にいえば，言語によるハンデを生じさせないという方針がとことん徹底されているわけではない．たとえば，C++ の STL（standard template library）を使えば，ソートやハッシュなどは簡単に実現できる．自分でプログラムを書く必要はない．これは，たとえば標準の Pascal と大きく違う．しかし，STL の使用を禁止することはもはや無理である．自分でプログラムを書く能力と上で言ったが，その中身は昔のプログラミング能力と少し違ってきているのかもしれない．

■水平線で薄切りにする方法

この問題の標準的な解法は，x 軸に平行な直線で図形を切る方法であろう．y を整数として，y 座標が y の線と，y+1 の線で，それぞれ図形を切る．すると，**図 -2** のように，辺の一部だけ見えるような図形ができるはずである．

このとき，y と y+1 の間の小正方形のうち，どれとどれが多角形と共通部分を持つかは簡単に調べられる．図形の下端から上端まで，y を 1 つずつ増やしながら，この操作を繰り返せば，問題の解が得られる．

多角形と共通部分を持つ正方形がどれであるかを調べるにはどうしたらよいだろうか．ちょっと幾何学的な考察を加えると，次のようなことが分かる．

> y と y+1 の間を走る辺のうち，左から奇数番目のものと偶数番目のものの間が多角形の内部である．逆に偶数番目の辺と奇数番目の辺の間は多角形の外部である．

y と y+1 の間を走る辺の本数は必ず偶数である．頂点の座標は整数なので，それぞれの辺は，y と y+1 の間を走っているか，それともこの区間の外にあるかのどちらかである．区間の途中まで入り込むなどということはない．

正方形の数を求めるには，奇数番目の辺と偶数番目の辺がそれぞれ上下の水平線と交わる点の x 座標を求めればよい．計 4 つの交点ができるが，このうち最も左の点から最も右の点までが 2 つの辺にはさまれる区間である．正方形が多角形と少しでも重なりを持てばよいという条件を正確に表現するために，左端の交点の x 座標を切り下げ（切り捨て），右端の交点の x 座標を切り上げる．こうして得られた 2 つの値の差

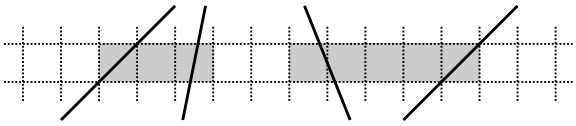

図-2　水平線による薄切り

がこの 2 本の辺ではさまれる部分の求める正方形の個数である．たとえば，図 -2 の左側 2 本の辺の水平線との交点の x 座標が 5.0，6.0，7.4，7.6 だとしよう．4 つのうちの最小値 5.0 を切り下げて得られる整数値は 5 である．また，最大値 7.6 を切り上げて得られる整数値は 8 である．8−5=3 が正方形の個数になる．図を見れば，このとおりであることが容易に納得できるだろう．

■プログラムとしての実現

水平線薄切り法のプログラムとしての実現を考えよう．最初にやるべきことは，y 座標の最小値，最大値の計算である．たとえば，次のようにすればよい．

```
ymin = +2000;
ymax = -2000;
for (i = 0; i < n; i++) {
  if (a[i].y < ymin)
    ymin = a[i].y;
  if (a[i].y > ymax)
    ymax = a[i].y;
}
```

この後の処理の概略は次のようになる．

```
area = 0;
for (y = ymin; y < ymax; y++) {
  ・ n本の辺のうち，yとy+1ではさまれる区間を走るものだけを集め，それぞれの上下の水平線との交点のx座標を求める．
  ・ そのx座標に基づいて，辺が左から順になるようにソートする．
  ・ 上に示した計算法で正方形の個数を求め，areaの値をその分だけ増やす．
}
```

最初の処理（この区間を走る辺を集める）は次のとおりである．

```
k = 0;
for (i = 0; i < n; i++) {
  if ((a[i].y <= y && a[i+1].y <= y) ||
      (a[i].y >= y+1 && a[i+1].y >= y+1))
    continue;
  compute_x(i, y, &xla, &xra);
  compute_x(i, y+1, &xlb, &xrb);
  edge[k].left = min(xla, xlb);
  edge[k].right = max(xra, xrb);
  k++;
}
```

ここで，配列 edge は次のように宣言してあるとする．メンバ left には上下の交点のうちの小さい方の切下げ値，right には大きい方の切上げ値を入れる．

```
struct { int left, right; }
  edge[Max_Vertex];
```

max と min は，2 つの値の大きい方，小さい方をそれぞれ返す関数であるが，次のようにマクロとして定義するというテクニックがよく使われる．

```
#define max(x, y)  ((x) > (y) ? (x) : (y))
#define min(x, y)  ((x) < (y) ? (x) : (y))
```

compute_x は，辺の番号 *i* と水平線の座標 *y* を受け取って，交点の *x* 座標を求める関数である．交点の *x* 座標は一般に実数であるが，それを切り下げした値と切り上げした値を求め，それぞれ引数 xl，xr を通して返すことにしている．もし交点の座標が整数なら，2 つの引数を通して返される値は同じである．

```
void compute_x(int i, int y,
               int *xl, int *xr)
{
  int  den, num, w;

  den = a[i+1].y-a[i].y;
  num = (y-a[i].y)*(a[i+1].x-a[i].x);
  w = a[i].x+num/den;
  if (num%den == 0) {
    *xl = w;
    *xr = w;
  }
  else if ((num%den)*den < 0) {
    *xl = w-1;
    *xr = w;
  }
  else {
    *xl = w;
    *xr = w+1;
  }
}
```

第 2 の処理（辺のソート）は簡単に済ませよう．せいぜい 100 本しかないのだから，高級なソートアルゴリズムを使うまでもない．

```
for (i = 0; i < k; i++)
  for (j = i+1; j < k; j++)
    if ((edge[i].left > edge[j].left) ||
        (edge[i].left == edge[j].left &&
         edge[i].right > edge[j].right)) {
      tmp = edge[i];
      edge[i] = edge[j];
      edge[j] = tmp;
    }
```

馬鹿ソートで十分である．2 つの辺の比較は，それぞれの左端の座標を先に比べ，それが同じ場合は右端の座標を比べることにしている．

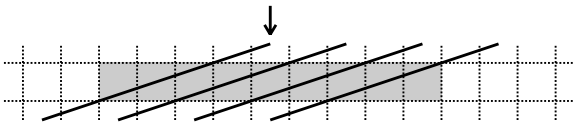

図-3　水平に近い辺が多い図形の場合

ここまでできれば後は簡単である．次のようなプログラムで面積が求まる．

```
prev = -2000;
for (i = 0; i < k; i += 2) {
  area += edge[i+1].right
        - max(edge[i].left, prev);
  prev = edge[i+1].right;
}
```

ここで，変数 prev に注意してほしい．これは辺の対（奇数番と偶数番）の右端の座標を記憶するための変数である．その次の対の占める正方形の数を計算するとき，単純に右端から左端を引くのではなく，max(左端，prev) を右端から引いている．これは，**図 -3** のような図形があり得るからである．

左側の 2 本の辺ではさまれる部分とも右側の 2 本の辺ではさまれる部分とも共通部分を持つ正方形が存在する（図の矢印で示した正方形）．このような場合，単純にそれぞれの辺の対についての右端マイナス左端の値を加えていくだけでは，同じ正方形を 2 回数えることになって，最終的な答が違ってしまう．変数 prev は，一度数えた正方形を二度と数えないようにするための工夫である．

■陥りやすい罠

水平線薄切り法のプログラムは以上で終わりである．大したコード量ではないし，一見したところ難しい処理をしているとも思えない．最初に示した解法さえ発見できれば，自明なプログラミングしかないのだろうか．実は，話はそれほど簡単ではない．うっかりすると陥りかねない罠が隠されているのである．

罠の 1 つは，座標を整数に直すところにある．たとえば，切下げの場合を考えてみよう．与えられた値（実数）以下で最大の整数を求めることが目標である．元の値は，2 つの整数 a と b の商の形で与えられるから，単純に考えれば a/b とするだけでよいように見える．整数の割算が切捨てをしてくれるからである．

ところが，これが間違いである．割算の切捨ては，0 に近づく方向に丸めると規定されている．たとえば，(−3)/2 は −1 になる．これは，−1.5 以下で最大の整

数とは違う．本来求めたい値は −2 である．

問題なのは，被除数を動かしたとき，0 を境に割算の振る舞いが不連続になる（余りの符号が逆転する）ことである．上のプログラムでは，余りの符号による場合分けをして，これによる悪影響を避けている．本当は，被除数が正であろうと，負であろうと，除数の符号で余りの符号が決まるようになっていればプログラミングが楽になるのだが，ハードウェアがそうなっていない以上，プログラミング言語に期待することは難しいだろう．自分でプログラミングするしかない．

上のプログラムでは，座標値に対する計算をすべて整数で行っている．現れる数が −2000 ～ +2000 の範囲に限られているから，実数計算にしても，大した誤差は生じず，正しく計算できるように見える．

しかし，これも正しくない．たとえば，筆者の計算機で 2000 ÷ 3999 × 3999 の計算を実数で行ってみたところ，結果が 2000 ピッタリにはならなかった．ほんの少しの誤差が生じて，2000 よりわずかに大きな値になる．普通の計算なら，この程度の誤差があってもどうということはないのだが，この問題の場合は困る．切り上げた結果の値が正しくなくなるからである．正解の 2000 なら切上げ結果も 2000 だが，それよりわずかに大きな値の切上げ結果は 2001 となって，違いが生じる．この違いは，最終的な答の間違いとなって現れるから，小さな誤差といって無視することはできない．

水平線薄切り法を振り返って見ると，幾何的な考察が難しいとはいえないと思う．プログラムも，さほどの量にはならず，使っている技法も難しいものではない．したがって，問題自体の評価として，難問ということはできないと思う．

しかし，実際のコンテストでは，34 チーム中 3 チームしかこの問題を解けなかった．挑戦したが解けなかったチームも 2 つしかなく，学生から敬遠された格好である．とっつきにくいと思われたのだろうか．筆者は，幾何的考察とプログラミングの両方の能力を問う面白い問題だと考えていただけに，この結果は残念であった．

■塗りつぶしによる解法

この問題の解法としては，水平線薄切り法が決定版だと思われる．これに勝る簡単な解法があるとは思えない．しかし，解き方が 1 つしかないのでは面白くない．無理にでも別解を見つけてやろうとひねり出したのがここに示す方法である．

基本的な考え方は単純である．発想としては，こちらの方が素直かもしれない．最初に，多角形の辺が通る正方形に印を付ける．すると，多角形内部は，印の付いた正方形に囲まれたような状態になる．この状態から，閉領域の塗りつぶしの要領で，内側の正方形に次々に印を付けていく．塗りつぶしが終わった時点で，印の付いた正方形の数を数えればよい．

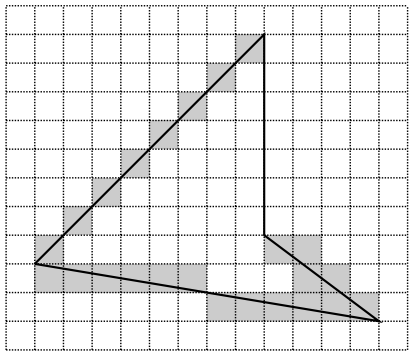

図-4　鉛直な辺がある場合

閉領域の塗りつぶしは，非常に簡単である．次のような関数で実現できる．この関数の書き方を覚えておいて損はない．配列 mark の値によって，各正方形に印が付いているか（True），否か（False）を表すとする．

```
void paint(int x, int y)
{
  if (mark[x][y] == True)
    return;
  mark[x][y] = True;
  paint(x+1, y);
  paint(x-1, y);
  paint(x, y+1);
  paint(x, y-1);
}
```

これだけで元の問題が解ければよいのだが，残念ながら話はそれほど簡単ではない．この方法で問題を解くまでには，越えなければいけない山がいくつもある．主なものは次の 3 つである．

(1) 水平または鉛直な辺

辺が通る正方形に印を付けた後の状態は，たとえば**図 -4** のようになる．これで分かるとおり，辺の中に水平なものや鉛直なものがあると，その辺のところだけ正方形に印が付かず，塗りつぶしができない（図形の外にはみ出す）状態になる．

この問題は，辺に接する正方形に特別な印を付けることで解決できる．たとえば，図 -4 のような鉛直な辺の場合，辺のすぐ左の正方形には右への移動禁止，すぐ右の正方形には左への移動禁止の印を付ける．上の paint のようなプログラムで，移動禁止の方向には行かないようにすれば，辺を越えて外に出ることは起こらず，塗りつぶしが成立する．

(2) 多角形外部の閉領域

図 -5 の場合，印付きの正方形で囲まれた領域が 3

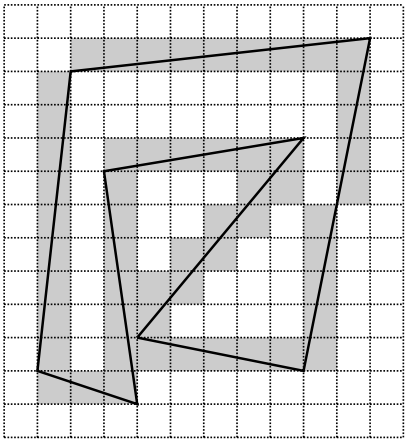

図-5　多角形外部の閉領域

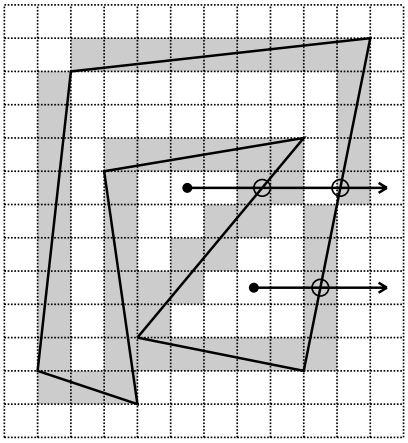

図-6　多角形の中か外かの判定

つできるが，このうちの 1 つは多角形の外である．この領域に対して無造作に塗りつぶしを行ってはならない．つまり，塗りつぶしを始める前に，その領域が多角形の中であるか外であるかの判定をしなければならない．

ある点が多角形の中にあるか外にあるかは，その点から x の正方向に直線を引き，いくつの点で辺と交わるかを求めれば判定できる．辺との交点が偶数個なら多角形の外，奇数個なら多角形の中である（**図 -6**）．

(3) メモリ量の問題

上に示した関数 paint が行っていることは，深さ優先探索である．先へ進めるだけ進んで，行き止まりになったら引き返す．たとえば，塗りつぶし対象として 4000 × 4000 の正方形（問題で許される最大の図形）が与えられたらどうなるだろうか．1 列ごとに行ったり来たりする格好で，1600 万個の正方形を 1 列につないだ線の上をドンドン進むはずである．つまり，上のプログラムでは，1600 万段の再帰呼出しが起こる．これではたまらない．メモリが足りなくなって，プログラムが異常終了してしまう．

再帰呼出しをやめて，スタックを使った繰返しにすればメモリ量を減らせる（筆者の環境では，再帰呼出し 1 段あたり 112 バイトのメモリを使っているらしい）が，深さ優先探索である限り大量のメモリを必要とすることに変わりはない．この場合の解決策は，深さ優先探索をやめて，幅優先探索にすることである．幅優先探索なら，図形の最大幅（この問題の場合は 4000）程度の個数のデータを記憶できれば十分である．

■プログラムとしての実現

塗りつぶし法のプログラムとしての実現に移る．初めに，図形全体を平行移動して，すべての頂点の座標値が 0 以上になるようにしよう．これは本質的ではないが，何かとプログラムが簡単になる．この結果，x 座標の範囲は 0 ～ xsize，y 座標は 0 ～ ysize になったとする．

次に，辺が通る正方形に印を付ける．上の (1) で述べた移動禁止の印は簡単なので，水平でも鉛直でもない辺の場合だけを示そう．i 番と i+1 番の頂点を結ぶ辺が通る正方形への印付けである．

```
if (a[i].y < a[i+1].y) {
  s = i;
  t = i+1;
}
else {
  s = i+1;
  t = i;
}
dx = a[t].x-a[s].x;
dy = a[t].y-a[s].y;
xa = a[s].x*dy;
for (y = a[s].y; y < a[t].y; y++) {
  xb = xa+dx;
  for (x = min(xa, xb)/dy;
       x < (max(xa, xb)+dy-1)/dy; x++)
    map[x][y] |= Inside;
  xa = xb;
}
```

変数 xa や xb に入っている x 座標の値は，真の値の dy 倍になっている．本来なら分数の分子に置くべき値を単独で取り出した格好である．

印付けは，配列 map の各要素の特定のビットを 1 にすることによって行う．移動禁止の印を含めると，このプログラムでは次の 6 種類の印を使っている．

```
#define Barrier_Down    0x01  /* 下移動禁止 */
#define Barrier_Up      0x02  /* 上移動禁止 */
#define Barrier_Left    0x04  /* 左移動禁止 */
#define Barrier_Right   0x08  /* 右移動禁止 */
#define Inside          0x10  /* 多角形内部 */
#define Outside         0x20  /* 多角形外部 */
```

多角形の外枠に印を付けることができたら，その後の仕事は次のとおりである．

```
int compute_area(void)
{
  int x, y, mark, area;

  for (x = 0; x < xsize; x++)
    for (y = 0; y < ysize; y++)
      if ((map[x][y]&(Inside|Outside))
            == 0) {
        if (is_inside(x+0.5, y+0.5))
          mark = Inside;
        else
          mark = Outside;
        set_mark(x, y, mark);
      }
  area = 0;
  for (x = 0; x < xsize; x++)
    for (y = 0; y < ysize; y++)
      if ((map[x][y]&Inside) != 0)
        area++;
  return (area);
}
```

印の付いていない各点について，多角形の中か外かを判定し，それぞれの印で印付けをする．多角形の外から印付けを始めると，無限に遠い所に行ってしまう可能性があるが，筆者のプログラムでは盤面全体の外枠に移動禁止の壁を作ることによって防いでいる．

多角形の中か外かを判定する関数は次のとおりである．上の compute_area を見ると分かるとおり，この関数に渡される座標値は，整数 +0.5 の実数値になっている．整数だと，頂点をかすめるようなケースを心配しなければならないが，整数と整数の真ん中なら面倒な事態が起こらない．

```
int is_inside(double x, double y)
{
  int count, i;
  double r, w;

  count = 0;
  for (i = 0; i < n; i++) {
    if (max(a[i].y, a[i+1].y) < y)
      continue;
    if (min(a[i].y, a[i+1].y) > y)
      continue;
    r = (double)(a[i+1].x-a[i].x)
      / (double)(a[i+1].y-a[i].y);
    w = r*(y-a[i].y)+a[i].x;
    if (w > x)
      count++;
  }
  if (count%2 != 0)
    return (True);
  else
    return (False);
}
```

ここまでできれば，後は簡単である．閉領域を印付けするプログラムを書くだけである．ただし，深さ優先探索でなく幅優先探索を行うようにしなければならないし，移動禁止の壁を越えて動くことがないようにするチェックも必要である．最初に示した paint より，だいぶ複雑なプログラムになった．しかし，よく見ると，やっている仕事の本質部分は paint と大して違わないことが分かるだろう．paint を理解していれば，このプログラムの理解も難しくない．

```
void set_mark(int x, int y, int mark)
{
  q_head = 0;
  q_tail = 0;
  q_count = 0;
  put_queue(x, y, mark);
  while (q_count > 0) {
    x = queue[q_tail].x;
    y = queue[q_tail].y;
    q_tail = (q_tail+1)%Max_Queue;
    q_count--;
    if ((map[x][y]&Barrier_Left) == 0)
      put_queue(x-1, y, mark);
    if ((map[x][y]&Barrier_Right) == 0)
      put_queue(x+1, y, mark);
    if ((map[x][y]&Barrier_Up) == 0)
      put_queue(x, y+1, mark);
    if ((map[x][y]&Barrier_Down) == 0)
      put_queue(x, y-1, mark);
  }
}

void put_queue(int x, int y, int mark)
{
  if (x < 0 || y < 0 ||
      x >= xsize || y >= ysize)
    return;
  if ((map[x][y]&(Inside|Outside)) != 0)
    return;
  map[x][y] |= mark;
  queue[q_head].x = x;
  queue[q_head].y = y;
  q_head = (q_head+1)%Max_Queue;
  q_count++;
}
```

深さ優先探索がスタックを使って実現できるのに対して，幅優先探索は待ち行列（キュー）を使うことによって実現できる．

以上で，塗りつぶしの方法で問題を解くことができたが，はじめに述べたとおり，これは無理矢理解いたという域を出ない．プログラムは長いし，計算時間も数倍余計にかかる．繰返しになるが，水平線薄切り法に勝る解法はなさそうである．

（平成 15 年 12 月 10 日受付）