円の集まりをロープで囲む

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

今回は2002年10月国内予選の問題E「Enclosing Circles」を取り上げる．この問題の説明に言葉は要らない．2つの図を見てもらえば，それだけで一目瞭然であろう．

図-1が入力，図-2が解として求めるべきものである．与えられた円すべてを取り囲むようなロープの張り方の中で，最も小さいもの，つまりピッタリ囲むものを求めればよい．そうして得られたロープの長さを答えるよう要求されている．

幾何の問題としては，それほど難しいものとも思えないが，実は誤差の扱いがなかなかやっかいである．素直に書いたプログラムでは，破綻するような嫌らしいデータが存在する（審判の判定データにこの種のデータは含まれていない）．破綻が起きないようなプログラムを頑張って書くことは当然可能なのだろうが，筆者にも一部解決できない問題が残っている．

この種の誤差の問題は，幾何的に厳密な解を求めようとすると，なかなか避けにくいようである．むしろ，最初から近似的に解を求めることに徹すれば，ずっと簡明で，誤差にも強いプログラムが得られる．厳密解法より近似解法の方が誤差に強いのは逆説的な話だが，こういうこともあるという例として見てもらえば，面白かろうと思う．

■問題

複数個の円が与えられている．それぞれの円についてデータとして与えられるのは，中心の座標 (x, y) と半径である．円はすべて同一平面上にあるが，それ以外の位置関係に特に制約はない．1つの円が別の円を完全に包含していることもあるし，円と円が一部重なっていることもある．

プログラムの入力は，次のような形式になる．

n
x_1 y_1 r_1
x_2 y_2 r_2
...
x_n y_n r_n

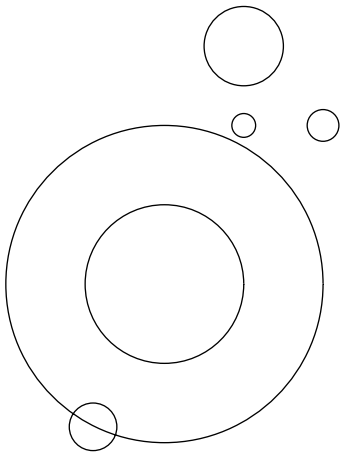

図-1　入力データ

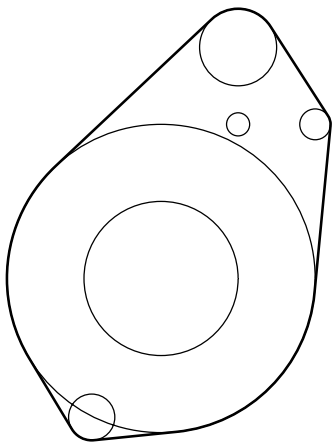

図-2　求める解

最初の行が円の個数 n である．n は 100 以下と定められている．2 行目以降がそれぞれの円のデータで，中心の x 座標，y 座標，半径の順になっている．入力に与えられる座標値や半径は，0.01 以上，100.0 以下の範囲内にある．

ほとんど同じ円が 2 つ以上あると，いろいろやっかいなので，そのようなことはないと保証されている．具体的には，2 つの円の x 座標，y 座標，半径のうちの少なくとも 1 つは 0.01 以上差があると規定されている．

出力は，ピッタリ囲んだときのロープの長さである．小数点以下 5 桁まで表示することが求められている．誤差は 0.00001 以下でなければならない．これは，0.00001 の倍数のうち，真の解の前後 2 つのどちらかを出力しなければならないことを意味する．

■凸包からの類推

この問題と似た問題に凸包（convex hull）がある．これは，点の集合を取り囲む最も小さな凸多角形を求める問題で，幾何の分野の代表的な問題の 1 つである．ここで取り上げた問題が凸包の解法と同様のやり方で解けそうなことはすぐに想像がつく．凸包には効率的なアルゴリズムがいくつも工夫されているので，それを利用すればよかろう．ただし，今回の問題の場合，計算量の小さなアルゴリズムを選ぶことにそれほどの意味はない．円の数が 100 個以下なので，計算量のオーダが少しくらい悪くても，大した時間はかからないからである．

凸包の解法からの類推で，すぐに思いつく解は次のようなものであろう．基本的な考え方は，ロープが走っている経路を忠実に再現することである．初めに，ロープに接していることが確実な円を 1 つ選ぶ．この円から出発して，時計回りにロープの走り方を追っていく．それぞれの円では，自分の次にロープが接する円を選んでそこに移動する．

出発点としては，右端の x 座標が最も大きい円を選べばよい．この円がロープに接していることは間違いないからである．x 座標最大の円が複数ある場合は，その点の y 座標（中心の y 座標に等しい）が最も小さいものを選ぶ．

図 -3 にこの解法をもう少し具体化したものを示す．右下の円を出発点に選んだとする．この円から他の円への共通接線は 3 本あるが，その中で最も左にあるものを選んで，左下の円に移動する．最も左の接線の

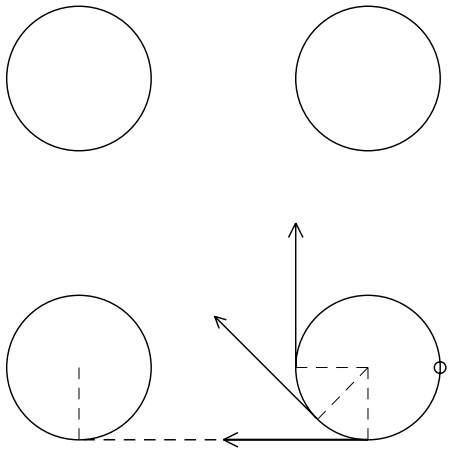

図 -3　円を順にたどる解法

上をロープが走ることは簡単に理解できるだろう．移動した先の円でも，同様に最も左の接線を選ぶ操作を行う．以下同様である．

最も左の接線を選ぶという部分をもう少し詳しく説明しよう．円を順にたどる操作では，どの円にいるかだけでなく，その円の中のどの位置にいるかも把握する必要がある．円の中の位置は，円の中心から見た角度（右が 0°または 360°，左が 180°）を使えばよい．

プログラムの最初では，右下の円の右側（図中の白丸の位置）にいる．角度は 360°である．この円の左側から出る接線は，それぞれ接点の角度で表現して，180°，225°，270°の 3 本ある．この中で最も大きい 270°から出る接線を選べばよい．このとき，移動した先の円の中の位置も同じ 270°であることに注意．接線と円の半径は，どちらの円でも直交しているからである．次の円では，270°から小さい方向に接線を探す操作を行う．

■プログラムとしての実現

入力データは，次の変数に読み込んであるものとする．

```
int  n;
struct { double x, y, r; }
   a[Max_Circle];
```

それぞれの変数およびメンバの意味は明らかだろう．`Max_Circle` は 100 を表す `#define` 定数である．

プログラムでは，初めに円と円の組合せすべてについて共通接線を求める．共通接線は円の外側を結ぶものと内側を交差する線で結ぶものの 2 種類があるが，

当然前者だけを考える．

この際，ある円の内側に完全に包含されている円があると話がややこしくなる．外側の円と内側の円の間には共通接線を引けないからである．包含されている円は，最終的な答に寄与しないはずだから，最初にデータから取り除いておくのがよい．この処理をするのが次に示す関数 remove_contained_circles である．

```
void remove_contained_circles(void)
{
  int  k, i, j;
  double d;

  k = 0;
  for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++) {
      if (i == j)
        continue;
      d = distance(a[i].x, a[i].y,
                   a[j].x, a[j].y);
      if (a[i].r <= a[j].r-d)
        break;
    }
    if (j >= n) {
      a[k] = a[i];
      k++;
    }
  }
  n = k;
}
```

ここでは，2つの点の間の距離を求める関数 distance を使っている．distance の定義は次のとおりである（ライブラリー関数 hypot が使える場合は，そちらを利用した方がよい）．

```
double distance(double x1, double y1,
                double x2, double y2)
{
  return (sqrt((x1-x2)*(x1-x2)+
               (y1-y2)*(y1-y2)));
}
```

共通接線の求め方は，**図-4** を使って説明しよう．左下の円 A から右上の円 B への接線を引くとする．まず，A の中心から B の中心に向かう線分の角度 θ は，2つの中心の座標差から逆正接関数（arctan）を使えば求められる．

これに加える角度 ϕ は，$\Delta r/d$ の逆余弦関数（arccos）で求まる．Δr は A の半径と B の半径の差，d は中心間の距離である．ただし，B の半径が A の半径より大きいとき ϕ は 90° より大きくなり，逆に B の半径が小さいとき ϕ は 90° より小さくなる．したがって，

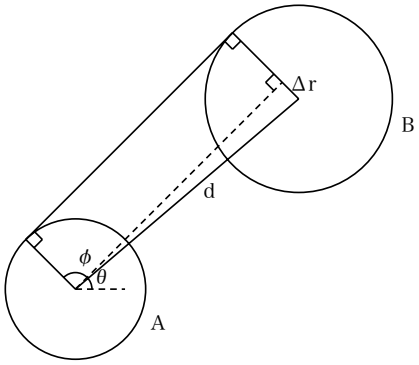

図-4　接線方向の計算

図中の記号とはちょっと矛盾するが，$\Delta r = r_A - r_B$ ととらなければならない．

共通接線の角度は，円の中心から接点に向かう法線の角度で表すことにしている．図の $\theta + \phi$ がこの角度にあたる．

この方法に基づいて，共通接線をすべて求める関数 connect_circles は次のとおりである．

```
void connect_circles(void)
{
  int  i, j;
  double d, t, s;

  for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++) {
      angle[i][j] = 1.0e+10;
      if (i == j)
        continue;
      d = distance(a[i].x, a[i].y,
                   a[j].x, a[j].y);
      t = acos((a[i].r-a[j].r)/d);
      s = atan2(a[j].y-a[i].y,
                a[j].x-a[i].x);
      angle[i][j] = s+t;
      if (angle[i][j] < 0.0)
        angle[i][j] += 2.0*M_PI;
    }
  }
}
```

求めた共通接線に関する情報は，配列 angle に記録する．angle の宣言は次のとおりである．

```
double angle[Max_Circle][Max_Circle];
```

配列 angle の各要素には，それぞれの共通接線の角度を入れる．共通接線を持たないときは，非常に大きな値を入れておく（こうしておけば，後の計算の邪魔にならない）．2つの添字のうち，左が自分の番号，

右が相手の円の番号である．

角度は0°～360°の範囲に収まるよう正規化しておかなければならない．arctanとarccosを足しただけでは負の値が得られる可能性がある．プログラムでは，角度の単位としてラジアンを使っているので，負ならば2πを加えるということをしている．M_PIは，円周率を表す定数である（標準ヘッダ<math.h>の中で定義されている）．

最後に，円を順にたどってループの長さを求める関数trace_pathは次のように書ける．

```
double trace_path(void)
{
  int  s, i, j, p;
  double t, new_t, max;
  double x1, x2, y1, y2;
  double length;

  max = 0.0;
  for (i = 0; i < n; i++)
    if (a[i].x+a[i].r > max ||
        (a[i].x+a[i].r == max &&
         a[i].y < a[s].y)) {
      s = i;
      max = a[s].x+a[s].r;
    }
  i = s;
  t = 2.0*M_PI;
  length = 0.0;
  for (;;) {
    p = -1;
    max = 0.0;
    for (j = 0; j < n; j++)
      if (angle[i][j] <= t &&
          angle[i][j] >= max) {
        p = j;
        max = angle[i][j];
      }
    if (p < 0) {
      length += t*a[s].r;
      return (length);
    }
    new_t = angle[i][p];
    length += (t-new_t)*a[i].r;
    t = new_t;
    x1 = a[i].x+a[i].r*cos(t);
    x2 = a[p].x+a[p].r*cos(t);
    y1 = a[i].y+a[i].r*sin(t);
    y2 = a[p].y+a[p].r*sin(t);
    length += distance(x1, y1, x2, y2);
    i = p;
  }
}
```

最初のfor文では，出発点となる円の番号sを求めている．解法の説明で述べたとおり，出発点は右端のx座標が最大の円である．このような円が複数あれば，

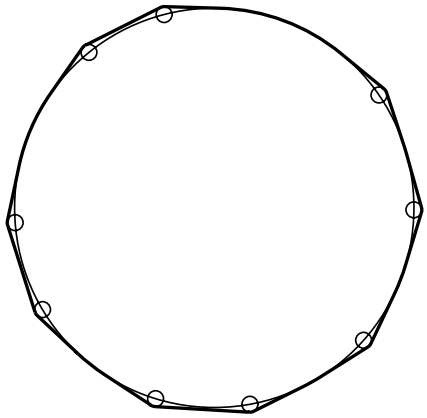

図-5　1つの円を何回も通るデータ

y座標が最小のものを選ぶ．

次の大きなfor文が円を順にたどる操作である．中心から見た接線の角度tが2πから0になるまで円をたどっていく．それぞれの円では，角度がt以下の範囲で最大の接線を探す．そして，得られた角度と行き先の円を次の計算に使う．「t以下の範囲」という条件は不要に思えるかもしれないが，必要である．図-5のようなデータがあり得るからである．

求めるループの長さは，変数lengthで管理している．接線の部分については線分の長さを加える．円弧の部分は，半径に角度を掛けたものを加える．角度の単位がラジアンなので，円弧の長さの計算は簡単である．

■誤差に対する配慮

最初に述べたとおり，この問題は計算中に生じる誤差への配慮が欠かせない．論理的には正しいプログラムでも，誤差に対する配慮を怠ると，とんでもない結果を生じることがある．

計算機における数値計算（実数計算）には常に誤差がつきまとう．数値を表現するビット数が有限なので，ある程度の誤差は覚悟していなければならない．これは常識である．どんな問題でも，これは避けられない．

しかし，ここで考える誤差の悪影響はこんなものではない．計算の仕方が悪いと，ほんのわずかの誤差のせいで，結果が大きく違ってしまうことがある．このような結果の違いは，普通の計算誤差よりもはるかに大きく，到底正しい結果とは認められないような値になってしまう．これは避けなければならない現象である．

たとえば，図-6のようなケースを考えてみよう．円の下から点Aに接線が入り，点Bから接線が出ていく状況を表している．円の内側から見て，2つの直線のなす角度は180°以下のはずだから，点Aの方が点Bよりも下にあるはずである．しかし，2つの直線の角度が180°にきわめて近い場合，数値計算の誤差によって，点Aの方が点Bより上になってしまったらどうなるだろうか．点Aに入った後，出ていく接線を探すが，この際Bから出る線は考慮に入らないので，本来とは違う円に行ってしまうことになる．

ポイントは，わずかな誤差のせいで，正解とはまったく違う経路をたどってしまうことがあるということである．図形のトポロジー的な認識を間違ってしまうと言い換えてもよい．当然ながら，こういう事態が起これば，正しい答は得られない．数値計算誤差とは比べものにならないくらい大きな違いとなって結果に現れるだろう．

計算幾何学の分野では，誤差のせいで起こるこの種の間違いに関する研究が盛んに行われている．一般に，誤差によって生じる悪影響を小さくとどめられるアルゴリズムのことをロバストなアルゴリズムと呼ぶ．

図-6に示した現象が実際に起こるかどうかは分からない．筆者が今までに実験した範囲では起こっていないから，杞憂だったかもしれない．しかし，上のプログラムがロバストでないことは確かである．次のようなデータを与えると，見事に間違えてしまう．

```
2
  1.0    1.000   31.830
 31.0    1.001    1.830
0
```

このデータで間違える原因は，角度が0～2πの範囲に収まるように調整する部分にある．2つの円の共通接線は，円の右端にきわめて近い点で接する．これを角度で表現すると，0よりほんの少し大きいか，2πよりほんの少し小さいかというところである．本来2π近くでなければならない値が誤差のせいで0近くになったら，やはり図形の認識を間違えてしまい，答は正しくないものになる．この現象が実際に起きているようだ．

この現象を起こさないようなプログラムを書くことは当然可能なはずである．しかし，プログラムが複雑になって，なかなか大変であろう．ここでは，そこまで頑張ることは避けたいと思う．

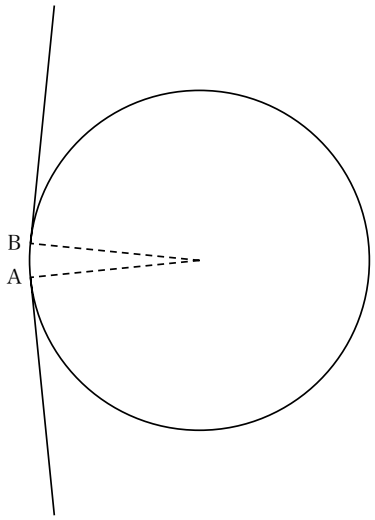

図-6　誤差の心配

■誤差に関するあれこれ

筆者のプログラムが間違えるのは，もちろんアルゴリズムがロバストでないことに原因がある．しかし，誤差に関しては，そのほかにも微妙な点がいろいろある．あまりまとまった話にはならないが，思いつくままいくつか紹介しよう．

問題では，入力がすべて小数点以下3桁の数値で与えられるとしている．つまり，入力としては0.001の整数倍だけを考えればよい．これを利用すれば，少なくとも入力値は誤差のない正確な値として読み取ることができる．普通のライブラリールーチンで実数を直接読み込まずに，小数点以上と小数点以下の2つに分けて，それぞれ整数として読み込めばよい．プログラム内部の計算は，すべての値を1000倍したままで行い，最後の結果の表示の直前で1/1000倍する．

この方法で実際にやってみると，前章の終わりに示したデータでも，エラーを起こさずに正しい結果を求められる．

これから分かるのは，入力値に含まれる誤差によって，ロバストでないアルゴリズムの弱点が端的に出てしまったということである．周知のように，浮動小数点数の2進表現では，0.1が正確には表現できない．また，たとえば31.830と1.830では，小数点以上の表現に必要なビット数が違うので，小数点以下の表現に使われるビット数が違う．同じ0.830であっても，若干違った値として表現されているはずである．このような小さな違いは，浮動小数点計算によって生じる誤差にとって一番の大敵であり，筆者のプログラムの

欠陥を浮き出させる結果になったと考えられる．

もちろん，整数で入力したとしても，筆者のプログラムが完全でないことに変わりはない．破綻させるようなデータを用意することは可能かもしれない．しかし，そう簡単なことではないだろう．

コンテストの問題を作成する過程で，審判団のメンバ何人かが実際にプログラムを作ってみて，問題文は理解しやすいか，データは正確か，想定されるアルゴリズムはどんなものか，など，さまざまな観点からの検討を加える．この問題の場合，審判が作ったプログラムの 1 つが不思議な振る舞いを示した．使用する計算機が Pentium なら正しい結果になるのに，Sparc だと間違えるというデータが 1 つあったのである．両者とも同じ浮動小数点数表現（いわゆる IEEE 形式，正確には IEC 60559）を使っているにもかかわらずである．

この原因は，Pentium の浮動小数点レジスターが 64 ビットを超える長さを持っていて，計算途中の数値を Sparc よりも高い精度で保持できることにあった．それが証拠に，コンパイラオプションによって，浮動小数点レジスタの拡張部分の使用を禁止してみたところ，Pentium でも Sparc と同様に間違えるようになった．

より本質的には，プログラム自体の論理に虫があることが原因だったようだが，とにかく浮動小数点数の計算では，このような意外なことがいくらでも起こり得る．十分に心しておかなければならない．

■近似解法

第 2 の解法として，近似的に解を求める方法を考えてみよう．これなら誤差によるトラブルは心配しなくて済む．本物と近似の差によって生ずる誤差は当然覚悟しなければならないが，近似の精度を上げることによって望むだけ小さくできるので，特に問題視する必要はないだろう．

図 -7 を見てほしい．図形全体の外側の，ある方向から直線を近づけていく．どれかの円に接したところで接近をやめ，接した点を記録する．このような操作を 360° のあらゆる方向から行えば，たくさんの接点が得られる．これらを集めてみれば，求めるループの軌跡を近似したような図形になっているはずである．

プログラムでは，360° を N 等分して，合計 N とおりの方向から直線を近づけて接点を求める操作を行う．k 番目の接点を P_k（$0 \leqq k \leqq N-1$）としたとき，P_k と P_{k+1} の間の距離を $0 \leqq k \leqq N$-1 の範囲ですべて加え合わせれば，求めるループの長さが計算できる．円なので，P_N=P_0 としておかなければならないことに注意．

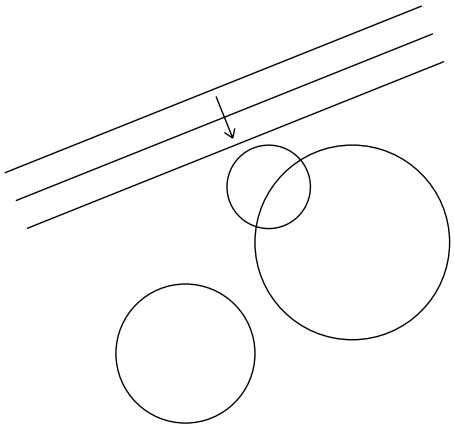

図 -7　近似解法

P_k と P_{k+1} の間の距離は，2 つの場合に分けて計算する．2 つの点が違う円に属している場合は，両者の間の直線距離を求める．同じ円に属している場合は，2 つの点を結ぶ線はその円の円弧になる．したがって，円弧の長さを両者の間の距離とすればよい．

■近似解法のプログラム

近似解法を実現した関数 approximate_path_length は次のとおりである．見てすぐ分かるように，最初の解法よりもはるかに短く簡単なプログラムになっている．

```
double approximate_path_length(void)
{
  double x[Partition+1], y[Partition+1];
  int  circle_id[Partition+1];
  int  k, i;
  double unit_angle, angle, c, s;
  double max, w, length;

  unit_angle = 2.0*M_PI/Partition;
  for (k = 0; k < Partition; k++) {
    angle = unit_angle*k;
    c = cos(angle);
    s = sin(angle);
    max = -1.0e+10;
    for (i = 0; i < n; i++) {
      w = c*a[i].x+s*a[i].y+a[i].r;
      if (w > max) {
        max = w;
        x[k] = a[i].x+c*a[i].r;
        y[k] = a[i].y+s*a[i].r;
        circle_id[k] = i;
      }
```

```
    }
  }
  x[Partition] = x[0];
  y[Partition] = y[0];
  circle_id[Partition] = circle_id[0];
  length = 0.0;
  for (k = 0; k < Partition; k++) {
    if (circle_id[k] == circle_id[k+1])
      length += a[circle_id[k]].r*
                unit_angle;
    else
      length += distance(x[k], y[k],
                         x[k+1], y[k+1]);
  }
  return (length);
}
```

Partitionは，360°を何分割するかを表す#define定数である．筆者のプログラムでは10000と定義してある．

最初のfor文で，それぞれの角度で直線を近づけたときの接点を求めている．x[k]とy[k]は接点の座標，circle_id[k]はその接点がどの円に属しているかを表す番号である．どの円に最初にぶつかるかの計算は，それほど難しくないので少し考えれば理解できると思う．ここでは説明を省略する．

なお，Partition番のデータに0番のデータをコピーしている部分を無駄に感じる人もいると思う．論理的には，最初のfor文を

```
for (k = 0; k < Partition; k++) {
```

から

```
for (k = 0; k <= Partition; k++) {
```

に変えれば，コピーは不要である．しかし，これはうまくいかない．2πが必ずしも正確に表現できないので，0のときの値と2πのときの値に微妙な差が生じることがあり，結果としてプログラム全体がロバストでなくなってしまう．

最後のfor文で，求めた接点の隣同士の距離を求め，すべて加え合わせてロープの長さを計算している．上に説明したように，隣同士が同じ円に属している場合と，それ以外の場合とに分けて，違う計算法を採っていることに注意．

■近似解法の精度

近似解法でどの程度の精度が得られるか確認しておくべきだろう．審判データに対する計算を行ったときの最大の誤差を表のかたちで示す．

分割数	円弧による近似	線分による近似
2500	1149	132392
5000	201	33115
10000	192	8264
20000	42	2064

第1のプログラムと第2のプログラムの計算結果の差を示してある．これを近似による誤差と考えてよいだろう．誤差は，1.0×10^{-9}の何倍になるかで示してある．たとえば，1149とあるのは，実際には0.000001149の誤差である．

分割数は，360°の方向をいくつに分割したかを示す．「円弧による近似」は，上の説明どおり，2点間の距離の計算に直線距離と円弧距離を使い分けた場合である．「線分による近似」は，円弧距離を使わず，直線距離だけを使って2点間の距離を計算した場合である．

これを見て分かるとおり，円弧による近似を使えば，分割数2500程度でも問題の要求する精度が得られている．問題文では，誤差0.00001以下を要求しているが，出力のときの四捨五入で最大0.000005の誤差が生ずるので，計算自体の誤差は0.000005より小さくなければならない．1149は十分この要求を満たしている．

円弧による近似を使わないと，分割数10000でも問題の要求を満たすことができない．20000くらいにして初めて満足のできる解となる．

近似解法の計算時間は，分割数20000とした場合に筆者のやや古い計算機で約4秒である．もちろん，最初の解法なら一瞬で計算が終わるので，それに比べれば遅いが，十分耐えられる時間であろう．

結論として，この問題を解くプログラムとしては，近似解法の方が決定版であると考えてよさそうである．

コンテストの際，この問題を解いたチームはなかった．さらに言えば，でき上がったプログラムを送ってきたチームすらなかった．出題者はそれほど難しい問題とは思っていなかったのだが，上に述べたようなもろもろの事情を勘案すると，かなり難しいと評価すべき問題だったようだ．

（平成15年8月17日受付）