点の集合を包含する球

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

今回は，幾何の問題を取り上げる．3 次元空間内の位置が与えられた複数の点すべてを包含する最小の球を求める問題である．2001 年 11 月の函館大会で問題 H として出題された．

■幾何の問題に関する一般論

プログラミングコンテストの問題は，数多くの分野から出題される．これは，いろいろな得意分野を持つ学生たちを公平に競わせるという観点から，意図的に行われていることであろう．数値計算，組合せ探索，グラフ，テキスト処理，…などさまざまなテーマの問題が出題される．

これら数多く存在する分野の中でも，幾何は特に重要な分野の 1 つである．どのコンテストでも 1 問や 2 問は必ず出題されている．実際に数えたことはないが，おそらく問題全体の 20% 近くは幾何の問題ではないかと思われる．

一口に幾何の問題といっても，問題の傾向はもちろんさまざまである．図形の種類だけでも，多角形を扱うもの，円を扱うもの，点の集合を扱うもの，…といろいろである．2 次元もあれば，3 次元もある．面積を求める問題，特定の性質を満たす経路を求める問題，何かの値の最大化または最小化を行う問題，…と計算の目的も数え切れない．

幾何の問題を解くときは，プログラミングの技法を考える前に，対象となる図形に関する数学的な考察を行わなければならないことが多い．たとえば，多角形の面積を求める問題を考えてみよう．いろいろな解き方があるが，多角形を三角形に分割した上で，三角形の面積の和として求める方法を思いつくことは難しくない．しかし，3 点の x-y 座標が与えられているとして，三角形の面積を求めるにはどうしたらよいだろうか．3 点の座標に対して面積の公式（底辺×高さ÷ 2）を直接適用できるはずもない．高校の数学でやったような式の計算を一生懸命やってからでないと，プログラムを書き始めることはできない．

実は，3 点の座標を (x_1, y_1), (x_2, y_2), (x_3, y_3) とすると，三角形の面積は次の式によって簡単に求まる．

$$S = \frac{1}{2}\left|(y_3-y_1)(x_2-x_1)-(y_2-y_1)(x_3-x_1)\right|$$

これなどは，知らなければとうてい導けない式だろう．座標に関する代数計算を必死でやれば得られることは確かだが，限られた時間のコンテストでそんなことをしている暇はないはずである．

上の式の絶対値記号の中の部分，$(y_3-y_1)(x_2-x_1)-(y_2-y_1)(x_3-x_1)$ には，三角形の面積計算以外の場面でも利用価値がある．特に，点の位置関係を判定する場合に利用できる点が重要である．この式の値は，3 点が反時計回り順に与えられているなら正，時計回りなら負になる．同じことを別の言い方で言うと，点 1 に立って点 2 の方向を見ているとき，点 3 が左側にあれば正，右側にあれば負になる．たとえば，点の集合の凸包（convex hull）を求める問題などで利用すると便利な式である．

結論として，幾何の問題を能率的に解くためには，この式を始めとする幾何計算の常識をいくつか身に付けておくべきである．コンテストでは参考書の持込みが許されているから，公式集に頼るという考え方も成立するが，その場合でも必要とする式をすぐに引き出せるよう練習しておく必要がある．

幾何問題のもう 1 つの特徴として，さまざまな解法が存在し得るということを挙げておきたい．いろいろな発想に基づいて，複数の方向からの攻略が可能な問題が多い．初等幾何の問題が補助線一発でスマートに解けるのと同じで，難しく見えた問題が，考え方を少し変えるだけで，いとも簡単に解けることもある．プログラミング問題としての面白さという点では一番であろう．

今回取り上げる問題も，常識的な解法のほかに，複

数の攻略法が存在する．そのうちの1つは，いかにも切れ味の鋭い見事な解法である．

■問題：点の集合を包含する球

3次元空間の中に n 個の点があって，これらの点の位置が与えられている．これらの点すべてを含む最小の球を求めるプログラムを書くことが問題である．球が点を含むという場合，点は球面の上に乗っていてもよいし，球の内部に位置するのでもよい．

点の個数 n は，4以上30以下である．点の位置は，ごく普通の x-y-z 座標で与えられる．x, y, z のいずれも0以上100以下の浮動小数点値である．また，点と点は少なくとも0.01以上離れていると規定されている．点の位置に関するこれらの制約は，プログラムを書く上で必ずしも必要ではないが，極端に数値的な性質の悪いデータはないと分かっているだけで安心感が生まれる．逐次近似を行う場合の初期値の設定や，誤差の見積りにも好都合である．

プログラムは，求めた球の半径を出力しなければならない．半径が求まっていれば，球の中心の位置も分かっているはずだが，こちらの出力は要求されていない．

半径の出力は，小数点以下5桁までと規定されている．また，真の値からの誤差が0.00001以下でなければならないとも要求されている．これは，実数値を答とする問題でよく使われる誤差の規定法である．小数点以下6桁目の四捨五入を間違える程度までは許すという意味を含んでいる．10進で7桁程度の精度があればよいのだから，普通にプログラムを書けば，誤差のせいで間違う恐れはほとんどない．特別に誤差が大きくなりやすい問題でもない限り，あまり誤差のことを気にする必要はないだろう．

以下では，入力した点の個数と位置が次のような変数に記録されていると仮定する．

```
typedef struct { double x, y, z; } pos;
int  n;
pos  point[30];
```

pos は，点の3次元座標値を記録するための型である．変数 n に点の個数，配列 point の point[0] ～ point[n-1] に各点の位置を記録する．

■代数方程式を解いて球の中心を求める方法

幾何の問題の例にもれず，この問題も数学的な考察なしではどこから手を付けたらよいかすら分からない．図形的な考察を少し行えば，次のような結論に到達することは難しくない．

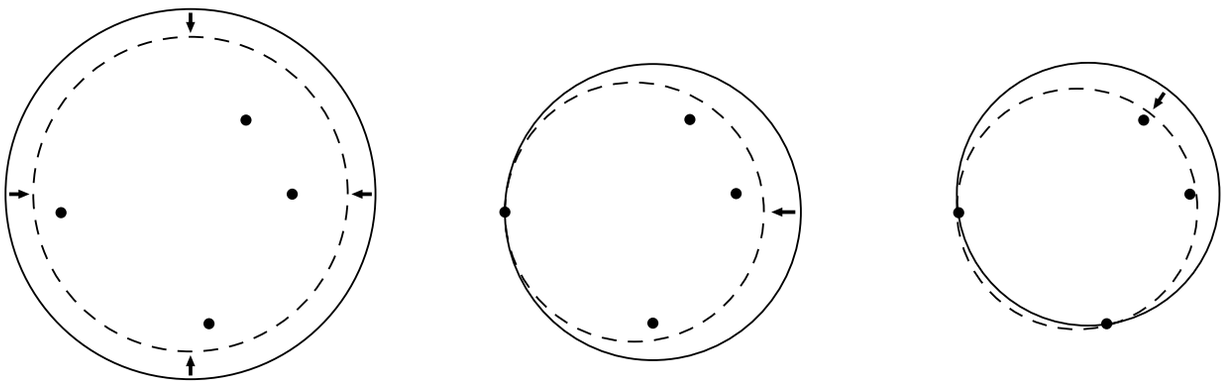
n 個の点すべてを包含する最小の球は次のいずれかである．

(1) n 個のうちの4個の点で作られる四面体に外接する球
(2) n 個のうちの3個の点で作られる三角形の外接円を大円とする球（三角形の外接円をその直径を軸として回転して得られる球）
(3) n 個のうちの2個の点を直径の両端とする球

この結論の正しさを確認する推論は次のようなものである．説明や図解を簡単にするために，ここでは2次元平面上の最小包含円を求める問題を対象にとって説明する．問題の本質は変わらないので，十分に類推できるはずである．

2次元の場合は，ケース(1)がなくなる．ケース(2)と(3)のいずれかであることを示せばよい．

(a) n 個の点すべてを包含する円の円周上に乗っている点が0個だったとする（図-1(a)）．この場合，円の半径を小さくしていくことによって，より小さい円が得られる．したがって，この円が最小であるはずはない．円の縮小は，n 個の点のいずれかにぶつかるまで続けることができる．
(b) 円周上の点が1個だったとする（図-1(b)）．この場合，その点を円周上に乗せたまま，円の中心をその点に近づけることによって，より小さい円が得られる．この場合も，この円が最小であるはずはない．
(c) 円周上の点が2個だったとする（図-1(c)）．2点が直径の両端であれば，それが最小である．そうでなければ，この2点を乗せたまま，円の中心を2点の中点に近づけることによって，より小さい円が得られる．円の中心はもともと2点の垂直2等分線の上にあったことに注意．
(d) 円周上の点が3個以上だったとする．この場合，円はそれらの点のうちの3個で作られる三角形の外接円になっている．



(a)　(b)　(c)

図-1

結局，(a) と (b) の場合は最小ではなく，(c) が最小なら (3) のケースに，(d) が最小なら (2) のケースに該当することが分かる．以上で，(2) と (3) のケースを尽くせば最小包含円が求まることを説明できた．

元の問題のプログラムの作り方に戻ろう．上の推論さえできれば，次のような単純明快なプログラムに到達することは難しくない．

- あらゆる点の組合せに対して (1)，(2)，(3) で決まる球を片端から求める．それらの球の中で，他のすべての点を含んでいて，かつ最小のものを見つける．

実際には，(1)，(2)，(3) の計算は包含球の中心位置の候補を与えるだけと割り切ってしまう方が簡明であろう．何はともあれ，それらの位置を中心とする包含球を作ってみて，半径最小のものを見つければ，それが求める最小包含球である．中心位置が与えられていれば，包含球を求める操作は簡単である．n 個の点の中で最も遠い点までの距離を半径とすればよい．

(1) で決まる球の中心の座標は，4 点からの距離が等しいという代数方程式を立てて，これを解けば求まる．未知数が x, y, z の 3 つ，方程式も 3 つの 3 元連立一次方程式である．(2) の方が複雑 (3 点と同一平面上にあるという式が必要) だが，こちらも 3 元連立一次方程式を解くことによって求められる．

(1) の球は，全部で ${}_nC_4$ 個ある．これらの球の中心それぞれについて，n 個の点までの距離を計算する必要がある．球の中心を求める作業の回数が $O(n^4)$ あり，距離の計算にそれぞれ $O(n)$ の計算量がかかるので，全体の計算量は $O(n^5)$ になる．(2) や (3) の計算量は，これよりオーダが低い．

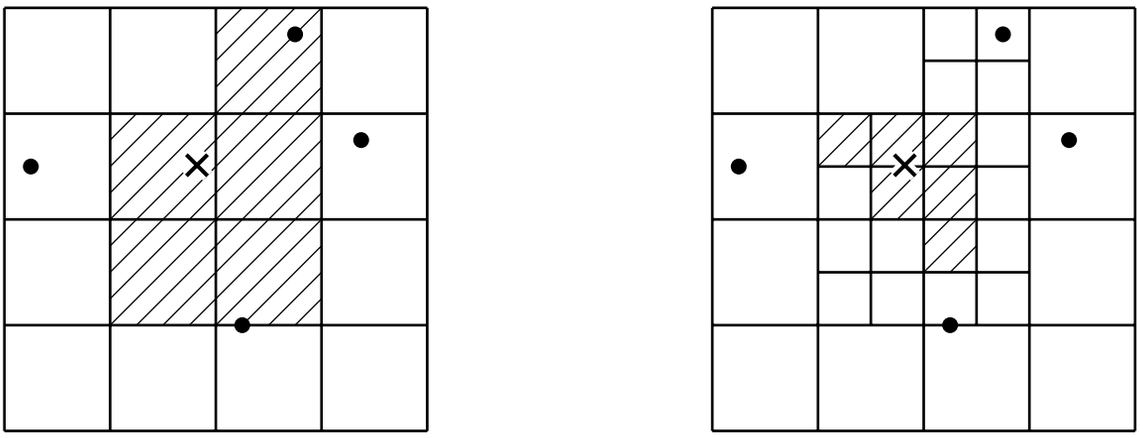
$O(n^5)$ の計算量では遅すぎる感じがするが，この問題の場合は，この程度でも十分間に合う．n の最大が 30 だからである．このアルゴリズムでも正解として許すよう n の上限を小さく定めてあるのだと想像できる．一般に，幾何の問題の場合，アルゴリズムの計算量を小さくすることはあまり重要でなく，解の求め方を図形的な考察によって構成することができればよしとしているものが多いようである．

この方式のプログラムはかなり長いものになるので，実際のプログラムを示すことは省略する．注意すべき点を 1 つだけ述べておこう．

4 点がたまたま同一平面上にあった場合，この 4 点で決まる四面体の外接球の中心は求まらない．不定または不能の方程式を解くかたちになる．具体的には，計算の途中で除数 0 の割り算が生じるはずである．同一直線上の 3 点があった場合も同様である．

プログラムには，これらの場合を検出する仕掛けを入れておかなければならない．これらの場合が本来の問題の解であることはないので，検出したら，その場合を単に無視するだけでよい．

最近の計算機は，浮動小数点型の値として無限大や NaN を扱える．これを利用すれば，同一平面上の 4 点などを特別扱いせずに済む可能性がある．不定または不能の方程式の場合は，球の中心として何でも (無限遠でもデタラメな位置でも) かまわないから，適当な点の座標を返してくれればよい．その点を中心とする包含球は最小のものより大きいはずだから，単に無視されるだけである．筆者は，このような考え方に基づいてプログラムを書いたのだが，これはうまくいかな



(a)　　(b)

図 -2

かった．数値計算ライブラリーの一部に NaN に十分に対応していないものがあるらしく，包含球の半径が 0.0 になってしまった．やはり特別扱いは必要である．

複数個の点を包含する円や球を求める問題は，計算幾何学の分野で十分に研究し尽くされた有名問題の 1 つである．ここに述べたのと同様の解法をアルゴリズム的に改良すると $O(n)$ にまで計算量を小さくできることが知られている．文献として次の 2 つを挙げておく．

1) ドバーグ他（浅野訳）：コンピュータジオメトリ，近代科学社 (2000).

2) Welzl, E.: Smallest Enclosing Disks (balls and ellipsoids), in H.Maurer (ed.) New Results and New Trends in Computer Science, Lecture Notes in Computer Science, Vol.555, pp.23-33 (1988).

■空間を分割する方法

幾何の問題は，正確に解こうとすると大変だが，近似的な解法なら簡単ということが多い．この問題の場合も，これが当てはまる．

幾何問題の近似解法としてよく使われる戦略に，空間を立方体の集まりとして把握し，それを順次分割していくものがある．立体図形の表現に使うデータ構造に octree（8 分木）と呼ばれるものがあるが，これと同様の方法である．

この問題の場合は，次のような考え方でこの方法を適用できる．

- プログラムでは，球の中心を含む可能性のある複数個の立方体を常に把握する．
- 最初に記憶する立方体は x，y，z 方向とも最小 0，最大 100 のもの 1 個だけである．
- 各立方体の中心から n 個の点までの距離の最大値を求める．その点を中心として n 個の点を包含する球を作ったときの半径である．これを r とする．立方体の 1 辺の長さを t とすると，立方体の中に中心を置いてできる包含球の半径が $r-\sqrt{3}t/2$ を下回ることはない．$\sqrt{3}t/2$ は，立方体の中心から 8 つの頂点までの距離である．
- 計算の進行に伴って，多数の立方体についての r を次々に求めていくが，それらの r の値の最小値を記憶する．これを r_{min} と呼ぶことにしよう．ある立方体について，$r-\sqrt{3}t/2$ が r_{min} を超えた場合は，その立方体を捨てる．最小包含球の中心がその立方体の中に存在する可能性がないからである．
- 捨てられずに残った立方体は，x，y，z の 3 つの方向に 2 等分して，合計 8 つの小立方体に分ける．そして，同様の操作を繰り返す．
- 立方体の大きさが誤差の限界を下回るまで小さくなったら，アルゴリズムを停止する．

図 -2 に，この方式を 2 次元で図解したものを示す．図で×印を付けた点が最小包含円の中心である．図 -2(a) では，平面全体を 16 の正方形に分けてあるが，このうち候補として生き残っているのは 5 つである（斜線を付けたもの）．これらの正方形をそれぞれ 4 分割（2 次元だから 8 分割でなく 4 分割）したものが図 -2(b)

である．この時点では，6つの正方形が候補として生き残っている．生き残りの正方形の分割を続けていけば，最小包含円の中心が存在し得る領域の面積をどんどん小さくすることができる．

この方式を実際のプログラムとして書いたものを示す．初めに，使用する型や大域変数の宣言である．

```
typedef struct { pos p; double r; } can;
can  candidate[Max_Candidate];
can  new_candidate[Max_Candidate];

int  sign[8][3] = {
    {  1,  1,  1 },
    {  1,  1, -1 },
    {  1, -1,  1 },
    {  1, -1, -1 },
    { -1,  1,  1 },
    { -1,  1, -1 },
    { -1, -1,  1 },
    { -1, -1, -1 } };
```

can は，候補立方体を記録するための型である．メンバ p は立方体の中心位置の座標，r はその点を中心としたときの包含球の半径を表す．

配列 candidate と new_candidate に候補立方体を記憶する．筆者のプログラムでは，配列の長さ Max_Candidate を 100000 とした．配列 sign は，8 等分を簡単に行うための補助配列である．

```
double solve_by_partition(void)
{
  int  count, k, i, d;
  pos  p;
  double size, r, root3, min_sofar;

  candidate[0].p.x = 50.0;
  candidate[0].p.y = 50.0;
  candidate[0].p.z = 50.0;
  count = 1;
  size = 100.0;
  min_sofar = 1.0e+50;
  root3 = sqrt(3.0);
  while (size > 0.000002) {
    size = size/2.0;
    k = 0;
    for (i = 0; i < count; i++) {
      for (d = 0; d < 8; d++) {
        p = candidate[i].p;
        p.x += sign[d][0]*size/2.0;
        p.y += sign[d][1]*size/2.0;
        p.z += sign[d][2]*size/2.0;
        r = compute_radius(p);
        if (r < min_sofar)
          min_sofar = r;
        if (r > min_sofar+size/2.0*root3)
          continue;
        if (k >= Max_Candidate)
          goto end_expand;
        new_candidate[k].p = p;
        new_candidate[k].r = r;
        k++;
      }
    }
  end_expand:
    count = k;
    for (i = 0; i < count; i++)
      candidate[i] = new_candidate[i];
    sort_candidate(count);
  }
  return (min_sofar);
}
```

while 文は，立方体の大きさを半分にする操作の繰返しを表す．for 文が二重になっているが，このうちの外側の方で候補立方体を順次調べていき，内側で 1 つの立方体を 8 つに分割する操作を行っている．立方体を分割したら，すぐにそれを候補として残すか捨てるかを判断している．compute_radius(p) は，点 p を中心とした包含球を作ったときの半径を求める関数である．

最初の方法に比べて，少なくともプログラミングはずっと楽である．こんな簡単な方法でも，大抵の場合は答が正しく求まる．

このプログラムの欠点は，候補立方体の数が多くなって，記憶域不足になる可能性があることである．最悪のケースは，2 点を直径の両端とする球が答である場合に生じる．2 点を結ぶ線分の中点から，この線分と直角の方向に少し移動した点は，2 点までの距離が真の中心とあまり変わらない．このような点を含む立方体は，$r-\sqrt{3}t/2$ の打切りではなかなか捨てられずに残る．線分の中点を中心とする円盤上にこのような立方体が多数存在するので，候補立方体の個数はすぐに数万のオーダを超え，億に達することもある．

候補の数がこのように爆発しても，元の問題が解けさえすればよい．これを可能にするために，上のプログラムでは，各ステップの最後の時点で，候補として残っている立方体をよさそうな順にソートしている．つまり，立方体の中心を中心としたときの包含球の半径（candidate[i].r の値）が小さいものほど配列の前の方に位置するように並べ替えをしている．これを担当するのが関数 sort_candidate(count) である．

分割の結果として，立方体の数が多くなりすぎた場

合は，その時点で分割操作を打ち切って，配列の後の方にある立方体を分割する操作は行わない．真の解は，おそらく配列の前の方にあるはずなので，打切りの前に処理が終わっているだろうと期待してのことである．

このほか，ここには示さないが，筆者のプログラムには候補数の爆発そのものを避けるための工夫も組み込んである．2つの点を直径の両端とする球が包含球になっているかどうかを調べる．このチェック自体は簡単である．包含球になっていれば，それが答なので，立方体分割の作業を始めずに直ちに結果を返す．

上に述べたとおり，候補の数が最も多くなるのは，直径の両端の2点だけが最小包含球の決定に貢献する場合である．この場合だけ特別扱いすれば，候補数爆発の可能性を大きく減らすことができる．

もちろん，この工夫だけで十分とは言えない．候補数の爆発が起こるデータはいくらでも作れる．いったん爆発が起こったら，何しろ計算の途中でデータを捨ててしまっているのだから，必ず正しい答が求まるとは保証できない．しかし，コンテストで使われた審判データ（爆発を起こす例も含まれている）で試してみたところ，候補の立方体の数が400あれば，すべての例について正解が得られた．候補の数を数万の程度にまで増やせば，間違える恐れはほとんどないと考えてよさそうである．

計算スピードは，あまり速いとは言えない．候補数を100000として，審判データで走らせると，コンテストの時間制限（3分）に引っ掛かるかどうか微妙なところである．候補数をこの半分くらいに減らせば安全であろう．

■球の中心に近づく操作を繰り返す方法

今度は，同じ近似解法でも，上のプログラムとはまったく違った発想に基づく方法を示そう．任意の位置から出発して，小さな移動を繰り返す（現在位置を次々と変えていく）ことによって，最小包含球の中心に順次近づいていくという方法である．空間内の1点を求めるための逐次近似法として最も自然な考え方であろう．たとえばNewton法を使って高次方程式の根を求めるのと同様の方法である．

ここまでは誰でも思いつくのだが，ハードルは高い．ポイントは，どのような規則に従って移動する先の位置を決めるかということである．最も単純な考え方としては，山登り法（hill climbing）があるが，これではうまくいかない．これは，現在位置の周囲で，より包含球の半径の小さい点を探して，そちらに移動するという方法である．半径が小さくなる方向を見つけることが大変なので，この方法はほとんど成立しない．場合によっては，きわめて小さな角度の方向に移動しないと半径を小さくできないことがある．

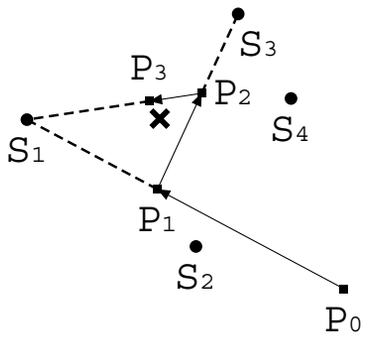


図-3

次に示すプログラムでは，現在位置の移動の規則として次のようなものを採用している．

- 各ステップでは，*n* 個の点の中で現在位置から最も遠い点の方向に向かって適当な距離だけ移動する．
- 移動距離は，ステップを繰り返すに従って徐々に小さくしていく．

だまされたように感じるかもしれないが，このような単純なルールによる移動で，確かに最小包含球の中心が求まる．図-3に2次元の例を示すので，これで納得してもらいたい．図で，$S_{1\sim4}$ は与えられた *n* 個の点を表す．$P_{0\sim3}$ が現在位置の推移である．P_0 から各点までの距離を計算すると S_1 が最遠方なので，S_1 の方向へ向かって動き，P_1 に移動する．ここで再び各点までの距離を計算すると S_3 が遠いので，その方向にある P_2 へ移動する．以下同様である．1回当たりの移動距離を徐々に小さくしていけば，求める最小包含球の中心（×印）に近づいていくことが見て取れると思う．

```
double solve_by_movement(void)
{
  int  k, i, t;
  pos  p;
  double move, max;

  p.x = 50.0;
  p.y = 50.0;
```

```
  p.z = 50.0;
  move = 0.5;
  while (move > 1.0e-8) {
    for (t = 0; t < 100; t++) {
      max = 0.0;
      for (i = 0; i < n; i++)
        if (distance(p, point[i]) > max) {
          max = distance(p, point[i]);
          k = i;
        }
      p.x += (point[k].x-p.x)*move;
      p.y += (point[k].y-p.y)*move;
      p.z += (point[k].z-p.z)*move;
    }
    move = move/2.0;
  }
  return (max);
}
```

変数 move は，最も遠い点までの距離に対する移動距離の比を表す．初めは，最も遠い点に向かって，その距離の半分（0.5 倍）だけ近づく．これを 100 回繰り返した後，比率を 0.25 にして，また 100 回繰り返す．以下同様で，移動距離の比を半分半分と小さくしていきながら，それぞれ 100 回ずつ移動を繰り返している．

0.5 とか 100 回とかは，いい加減に決めた値で，これが最善だという根拠は何もない．収束の具合に関する考察を加えることによって，もっと少ない繰返し回数で解に到達することは可能だと思われる．しかし，ここに示した方法でも十分に速いし，確実に答が得られる．細かな改良を加えることにあまり意味はないだろう．

この方法で最小包含球の中心を求められることの証明（現在位置と最小包含球の中心の間の距離が次第に小さくなっていく様子を示すことになるだろう）は結構大変である．しかし，直観的な理解でよければ，難しいことはない．その鍵になるのは，元の問題を一種の最適化問題として定式化することである．

問題で要求しているのは $\min_c \max_i distance(p_i, c)$ を求めることである．点 p_i と空間上の任意の点 c の距離を $distance(p_i, c)$ として，n 個の点の中の最大のものを選ぶと，これが包含球の半径になる．この最大値（半径）を最小にするような点 c が求める球の中心である．現在位置を最も遠い点の方向に移動すれば，$\max_i distance(p_i, c)$ が小さくなる．この操作を繰り返すことによって最小値に到達できることも，図形的に考えれば，ほぼ明らかである．

この方法は，プログラムの行数が最も少ないし，確実に答が得られる．実行時間もきわめて短い．データに依存して，悪い性質を示すということもなさそうである．この問題の解法として決定版と言ってよいだろう．

初めに述べたとおり，幾何問題には複数の解法が成立することが多い．包含球問題は，その好例である．幾何学的な考察に基づく厳密解法，空間を立方体に分けることによる近似解法，求める点に徐々に近づいていく近似解法と，バラエティに富む解法が可能であった．プログラミングの技法としてもそれぞれ面白い点を含んでいて，挑戦しがいのある問題だと思う．

■付言

Program Promenade の執筆者 4 人のうち，筆者だけがプログラミングコンテストの現役の審判である．

コンテストの審判団は，出題する問題それぞれについていろいろなプログラムを準備する．正解として想定するプログラム，アルゴリズムが悪くて遅すぎるプログラム，場合分けが不十分で正解にならないプログラム，等々である．これらのプログラムに基づいて，どんなデータを準備すればプログラムの正しさやスピードを検査できるか検討する．この作業の過程で，複数の審判から，いろいろな面白いプログラムのアイディアが提示されることが多い．

今回示したプログラムは，審判団の他のメンバが提案した解法を筆者なりの記法で書き直したものである．本当は，提案者である審判の実名を記して謝辞を述べるべきなのだが，審判団の内部事情を明かすことは好ましくないと思われる．ここでは，これらの事情を記録するだけにとどめたい．

（平成 14 年 8 月 13 日受付）