```cpp
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include <iostream>
using namespace std;
#include <fstream>

#define EVENT 100//ベント領域の一辺
#define EAREA 10201//EVENT*EVNT+(2*EVENT)+1
#define NODE 1000//ノード２０００個＋ＳＩＮＫ１個
#define hankei 40
#define hankei2 30
#define tyuusinti 200
#define RANGE 50//通信可能距離
#define hokanhankei 10//通信可能距離
#define Layernum 5//層の数(現状10層以内で作成)
#define kurasuta 10//クラスタの幅
#define RANSU 1000//乱数調整
#define Eelec 0.0000000001
#define kdeta 128//追加箇所
#define kheader 32//追加箇所
#define kid 96//追加箇所
#define kcontrol 20//追加箇所
#define omega 0.00000005//追加箇所




int main()
{

    int i, j, k, x, y, c, e, g, f, l, a, b;
    int cont = 0;
    int** link = new int*[NODE];
    int next = 0;
    int use[NODE];
    int now = 0;
    int next_hop[NODE];//シンクまでの最短経路における、次ホップ先のノード
    int** mark = new int*[NODE];
    int linkcount[NODE];//the number of link[NODE]
    int count = 0;
    int* cell = new int[100];
    int count0 = 1;
    int number = 0;//イベント内のノードの個数
    int edge_number = 0;//エッジノードの個数
    int edge_number2 = 0;
    int out_edge_number = 0;//未観測エッジノードの個数

    //srand((unsigned)time(NULL)); //ここを開くと乱数列が変わる




    int edge_no[NODE];
    int near = 0;
    int head = 0;
    int neighbor_node[NODE];//隣接ノードの数
    int neighbor_nonevent[NODE];//隣接未収集ノードの数

    int link_edge[NODE];//通信可能範囲内のエッジノードの数

    int* s = new int[NODE];//ノードの状態 (state) s[0]=未検出,s[1]=検出,s[2]=エッジ s
```

```
          [NODE][EVENT]=(EVENT)回目の時のノード(NODE)の状態
65      int* edge = new int[NODE];// (NODE)番ノードのエッジカウント
66      int** colle = new int*[NODE];
67      int* colle_number = new int[NODE];
68      int  saitan[NODE];//追加箇所
69      int  ruisekideta[NODE];
70      int  ruisekideta2 = 0;
71      int  cpu[NODE];
72      int  cpu1[NODE];
73      int  kansokuti2[NODE];
74      int  kansokuti3;
75      int  nara1[NODE];
76      int  nara2[NODE];
77      int  nara3[NODE];
78      int  nara4[NODE];
79      int  nara5[NODE];
80      int  nara6[NODE];
81      int max1 = 0.0;
82      int max2 = 0.0;
83      int min1 = 9999.0;
84      int min2 = 9999.0;
85      int minmin1 = 9999.0;
86      int minmin2 = 9999.0;
87
88      double menseki = 0.0;
89
90      int  nodonasi[EVENT];
91
92
93
94
95
96      double d = 0;
97      double** distance = new double*[NODE];//ノード間距離
98      double** distance2 = new double*[NODE];//ノード間距離の二乗
99      double** path = new double*[NODE];
100     double min = 1000;
101
102     double* distance_to_source = new double[NODE];//発生源とノード間の距離
103     double* distance_to_source1 = new double[NODE];//発生源とノード間の距離
104     double* distance_to_source2 = new double[NODE];//発生源とノード間の距離
105     double source_x = 0.0;
106     double source_y = 0.0;
107     double source_x1 = 0.0;
108     double source_y1 = 0.0;
109     double source_x2 = 0.0;
110     double source_y2 = 0.0;
111     double  kansokuti[NODE];
112     double  kansokuti1[NODE];
113     double  kansokuti4[NODE];
114     double radius = 0.0;//イベントの半径
115     double radius1 = 0.0;//イベントの半径
116     double new_radius = 0.0;//領域変化後イベントの半径
117
118     double neighbor_node_ave = 0.0;//隣接ノード数の平均
119
120
121
122
123
124     int max = 0;
125
126     int data = 0;
127
```

```
128
129
130         for (i = 0; i<NODE; i++)    //メモリ不足にならないように各配列の容量を確保
131         {
132             link[i] = new int[NODE];
133             mark[i] = new int[NODE];
134             distance[i] = new double[NODE];
135             distance2[i] = new double[NODE];
136             path[i] = new double[NODE];
137             colle[i] = new int[NODE];
138         }
139
140
141         //各ノードの座標を格納
142         struct zahyo
143         {
144             double x;
145             double y;
146         } node[NODE];
147
148
149
150
151         for (i = 0; i<NODE; i++)//初期化
152         {
153             for (j = 0; j<NODE; j++)
154             {
155                 distance[i][j] = 0.0;
156                 distance2[i][j] = 0.0;
157                 path[i][j] = 100000;
158                 link[i][j] = 0;
159                 mark[i][j] = 0;
160             }
161         }
162         for (i = 0; i<NODE; i++)//初期化
163         {
164             use[i] = 0;
165             next_hop[i] = 0;
166             linkcount[i] = 0;
167             distance_to_source[i] = 0.0;
168             distance_to_source1[i] = 0.0;
169             distance_to_source2[i] = 0.0;
170             neighbor_node[i] = 0;
171             neighbor_nonevent[i] = 0;
172             link_edge[i] = 0;
173             s[i] = 0;
174             saitan[i] = 0;
175             nara1[i] = 0;
176             cpu[i] = 0;
177             cpu1[i] = 0;
178         }
179
180
181
182
183
184         i = 0;
185         j = 0;
186
187         next = 0;
188         now = 0;
189         count = 0;
190
191
```

```
192
193
194     /*
195     //std::ofstream aoki( "result¥¥aoki.txt", std::ios::out | std::ios::app );//追記書き込み
196     std::ofstream aoki("result¥¥aoki.txt");//追記書き込みじゃない
197     //aoki<<"ノード"<<i<<"の要求データ量  "<<A.sdata[i]<<endl;
198
199     aoki << "ノード数 " << i << endl;
200     aoki << "通信範囲 " << endl;
201     aoki << "乱数調整 " << endl;
202     aoki << "ノード数" << endl;
203     aoki << "ノード数" << endl;
204     aoki << "ノード数" << endl;
205     */
206
207     //printf("ノード数 %d¥n", NODE);
208     //printf("通信範囲 %d¥n", RANGE);
209
210
211
212
213
214     i = 0;
215     j = 0;
216     k = 0;
217
218     for (i = 0; i<RANSU; i++)//乱数調整 ノードのランダムな座標配置のために必要
219     {
220         rand();
221     }
222
223
224
225     //=========================================================================ノードの座標配置
          ================================================================================
          ================================================//
226     //ノードの座標
227
228
229     for (i = 0; i<NODE; i++)// 配置されているか判定に使う
230     {
231         for (j = 0; j<NODE; j++)
232         {
233             mark[i][j] = 0;
234         }
235     }
236
237
238
239     node[0].x = 0.0;//SINKのx座標
240     node[0].y = 0.0;//SINKのy座標
241     mark[0][0] = 1;//SINKの位置は配置済み
242
243     for (i = 1; i < NODE; i++)//i:ノード番号
244     {
245         while (1)//ノードiが配置し終わるまで続ける
246         {
247             node[i].x = rand() % EVENT;//((k%(EVENT/10))*10)～((k%(EVENT/10))*10)+9
248             node[i].y = rand() % EVENT;//((k/(EVENT/10))*10)～((k/(EVENT/10))*10)+9
249             x = node[i].x;
250             y = node[i].y;
251             if (mark[x][y] != 0)//マーキングされていたら、配置しなおし
```

```
252             {
253                 continue;
254             }
255             mark[x][y] = 1;
256             break;
257         }
258 
259     }
260     for (i = 0; i < NODE; i++)
261     {
262         //printf("%f¥t%f¥n", node[i].x, node[i].y);
263     }
264     //=============================================================ノードの座標配置完了==========================================================//
265 
266     //=============================================================ノード間距離とリンク==========================================================//
267     for (i = 0; i<NODE; i++)
268     {
269         for (j = 0; j<NODE; j++)
270         {
271             distance[i][j] = sqrt((node[i].x - node[j].x)*(node[i].x - node[j].x) + (node[i].y - node[j].y)*(node[i].y - node[j].y));//三平方の定理
272             distance2[i][j] = distance[i][j] * distance[i][j];//ノード間距離の二乗の計算
273         }
274     }
275 
276     for (i = 0; i<NODE; i++)
277     {
278         for (j = 0; j<NODE; j++)
279         {
280             if (distance[i][j] <= RANGE && i != j)
281             {
282                 link[i][j] = 1;
283             }
284 
285         }
286     }
287     //=============================================================ノード間距離とリンク完了==========================================================//
288 
289     //=====================================================ダイクストラ==========================================================//
290     //参考URL:http://www.ss.cs.meiji.ac.jp/CCP024.html
291     //参考URL:http://www.deqnotes.net/acmicpc/dijkstra/
292     //printf("¥n=============ダイクストラ開始==============¥n");
293     now = 0;
294     path[0][0] = 0.0;//シンクのコストを0に
295     for (i = 1; i<NODE; i++)//他のノードを無限大に
296     {
297         use[i] = 0;
298         path[0][i] = 100000000;
299     }
300     use[0] = 1;//シンクを使用済みに
301 
302     while (1)//全てのノードが使用済みになるまで繰り返す
303     {
```

```
304         //printf("now=%d¥n",now);//現在地ノード
305         //printf("%f¥t%f¥n",node[now].x,node[now].y);//現在地ノードのxy座標
306         max = 10000000;
307         for (j = 0; j<NODE; j++)
308         {
309             if (link[now][j] == 1 && path[0][now] + distance[now][j]<path[0][j])//現在地ノードとノードjがリンクしている。かつ、ノードjまでの道のりが更新できる。
310             {
311                 path[0][j] = path[0][now] + distance[now][j];//最小コストの更新
312                 next_hop[j] = now;//シンクまでの最短経路における、ノードjの次経路。の更新
313             }
314             if (path[0][j]<max && use[j] == 0)//コストの最も小さい未使用のノードを次ノードとする
315             {
316                 next = j;
317                 max = path[0][j];
318             }
319         }
320         //printf("next=%d¥n",next);
321         if (now == next)//次ノードがなかったら
322         {
323             for (i = 1; i<NODE; i++)
324             {
325                 if (use[i] == 0)//もし未使用のノードがあったらシンクから探索再開
326                 {
327                     now = 0;
328                 }
329             }
330             if (now != 0)
331             {
332                 //printf("¥n==============探索完了==============¥n");
333                 break;
334             }
335         }
336         else
337         {
338             use[next] = 1;//次ノードを使用済みに
339             now = next;
340         }
341 
342     }
343     /*
344     for (i = 0; i<NODE; i++)//各ノード～シンク間の道のりの出力
345     {
346         printf("path[0][%d]=%f¥n", i, path[0][i]);
347     }
348     */
349 
350     /*
351     printf("¥n======================(NODE-1)番ノードからシンクまでの経路=======================¥n");
352     now = NODE - 1;
353     while (1)
354     {
355     printf("%f¥t%f¥n", node[now].x, node[now].y);
356     if (now == 0)
357     break;
358     else
359     now = next_hop[now];
360     }
361     */
362 
```

```
363     //=================================================ダイクストラ完了
          ================================================================================//
364
365     //=================================================イベントの発生とイベント観
          測値の設定
          ================================================================================//
366
367     //================イベントの発生源（中心点）=====================//
368     source_x = EVENT / 2;//ノード番号1125を中心に
369     source_y = EVENT / 2;
370
371     //================イベントの発生源とノード間の距離=====================//
372     for (i = 0; i<NODE; i++)//ノードiと発生源との距離
373     {
374         distance_to_source[i] = sqrt((node[i].x - source_x)*(node[i].x - source_x) +
              (node[i].y - source_y)*(node[i].y - source_y));//三平方の定理
375         //printf("distance_to_source[%d]=%f¥n",i,distance_to_source[i]);//ノードiと発
              生源との距離の出力
376     }
377
378
379     for (i = 0; i < NODE; i++)
380     {
381             kansokuti[i] = 0;
382     }
383
384
385     for (i = 0; i < NODE; i++)//ベンチマーク関数
386     {
387         kansokuti[i] = (1 + (node[i].x + node[i].y + 1) * (node[i].x + node[i].y + 1)
               * (19 - 14 * node[i].x + 3 * node[i].x * node[i].x - 14 * node[i].y + 6 *
              node[i].x * node[i].y + 3 * node[i].y * node[i].y)) * (30 + (2 * node[i].x
              - 3 * node[i].y) * (2 * node[i].x - 3 * node[i].y) * (18 - 32 * node[i].x +
               12 * node[i].x * node[i].x + 48 * node[i].y - 36 * node[i].x * node[i].y +
               27 * node[i].y * node[i].y));
388     }
389
390
391
392
393
394     /*
395     for (i = 0; i < NODE; i++)
396     {
397         if (distance_to_source[i] <= hankei)
398         {
399             kansokuti[i] = tyuusinti - distance_to_source[i] * ((tyuusinti - 10) /
                  hankei);//円錐
400         }
401     }
402     */
403
404
405     /*
406     for (i = 0; i < NODE; i++)//プリン型
407     {
408         if (distance_to_source[i] <= hankei)
409        {
410             if (distance_to_source[i] <= hankei2)
411             {
412                 kansokuti[i] = tyuusinti;
413             }
414             if (distance_to_source[i] > hankei2)
415             {
```

```
416                 kansokuti[i] = tyuusinti * 4 - (tyuusinti / 10) * distance_to_source
                      [i];
417             }
418         }
419     }
420
421     */
422
423
424
425     for (i = 0; i < NODE; i++)
426     {
427         printf("%f¥t%f¥t%f¥n", node[i].x, node[i].y, kansokuti[i]);
428     }
429
430     //=================================================イベントの発生とイベント観
          測値の設定完了
          ================================================================================//
431
432     //=================================================層の判定
          ================================================================================//
433     double layer = 0.0;
434     double layer1 = 0.0;
435     double layer2 = 0.0;
436     double layer3 = 0.0;
437     double layer4 = 0.0;
438     double layer5 = 0.0;
439     double layer6 = 0.0;
440     double layer7 = 0.0;
441     double layer8 = 0.0;
442     double layer9 = 0.0;
443     double layer10 = 0.0;
444     double layerhaba = 0.0;
445
446     layerhaba = tyuusinti / 20;//レイヤーの幅の決定
447
448     layer = (tyuusinti - 30) / Layernum;
449     layer1 = 10;
450     layer2 = layer * 1 + layer1;
451     layer3 = layer * 2 + layer1;
452     layer4 = layer * 3 + layer1;
453     layer5 = layer * 4 + layer1;
454     layer6 = layer * 5 + layer1;
455     layer7 = layer * 6 + layer1;
456     layer8 = layer * 7 + layer1;
457     layer9 = layer * 8 + layer1;
458     layer10 = layer * 9 + layer1;
459
460
461     for (i = 0; i < NODE; i++)
462     {
463         if (distance_to_source[i] > hankei)
464         {
465             s[i] = 0;//もしイベント範囲内なら５０　イベント外なら0　それ以外の数字は
                    層ナンバー
466         }
467     }
468     /*
469     for (i = 0; i < NODE; i++)
470     {
471         if (distance_to_source[i] <= hankei)
472         {
473             s[i] = 50;//もしイベント範囲内なら５０　イベント外なら0　それ以外の数字は
                    層ナンバー
```

```
474             }
475         }
476         */
477         for (i = 0; i < NODE; i++)
478         {
479             if (distance_to_source[i] <= hankei)
480             {
481                 s[i] = 1;//もしイベント範囲内なら５０　イベント外なら0　それ以外の数字は ➲
                    層ナンバー
482             }
483         }
484         for (i = 0; i < NODE; i++)
485         {
486             if (distance_to_source[i] < hankei2)
487             {
488                 s[i] = 2;//もしイベント範囲内なら５０　イベント外なら0　それ以外の数字は ➲
                    層ナンバー
489             }
490         }
491
492
493         /*
494         for (i = 0; i < NODE; i++)
495         {
496             if (kansokuti[i] > layer1)
497             {
498                 if (kansokuti[i] < layer1 + layerhaba)
499                 {
500                     s[i] = 1;
501                 }
502             }
503             if (kansokuti[i] > layer2)
504             {
505                 if (kansokuti[i] < layer2 + layerhaba)
506                 {
507                     s[i] = 2;
508                 }
509             }
510             if (kansokuti[i] > layer3)
511             {
512                 if (kansokuti[i] < layer3 + layerhaba)
513                 {
514                     s[i] = 3;
515                 }
516             }
517             if (kansokuti[i] > layer4)
518             {
519                 if (kansokuti[i] < layer4 + layerhaba)
520                 {
521                     s[i] = 4;
522                 }
523             }
524             if (kansokuti[i] > layer5)
525             {
526                 if (kansokuti[i] < layer5 + layerhaba)
527                 {
528                     s[i] = 5;
529                 }
530             }
531             if (kansokuti[i] > layer6)
532             {
533                 if (kansokuti[i] < layer6 + layerhaba)
534                 {
535                     s[i] = 6;
```

```
536             }
537         }
538         if (kansokuti[i] > layer7)
539         {
540             if (kansokuti[i] < layer7 + layerhaba)
541             {
542                 s[i] = 7;
543             }
544         }
545         if (kansokuti[i] > layer8)
546         {
547             if (kansokuti[i] < layer8 + layerhaba)
548             {
549                 s[i] = 8;
550             }
551         }
552         if (kansokuti[i] > layer9)
553         {
554             if (kansokuti[i] < layer9 + layerhaba)
555             {
556                 s[i] = 9;
557             }
558         }
559         if (kansokuti[i] > layer10)
560         {
561             if (kansokuti[i] < layer10 + layerhaba)
562             {
563                 s[i] = 10;
564             }
565         }
566     }
567
568     */
569
570     /*
571     for (i = 0; i < NODE; i++)
572     {
573         printf("%f¥t%f¥t%f¥t%d¥n", node[i].x, node[i].y, kansokuti[i], s[i]);
574     }
575     */
576     /*
577     for (i = 0; i < NODE; i++)
578     {
579         if (s[i] != 0 && s[i] < 10)
580         {
581             printf("%f¥t%f¥t%f¥t%d¥n", node[i].x, node[i].y, kansokuti[i], s[i]);
582         }
583     }
584     */
585
586
587     //==================================================層の判定完了
        ==========================================================================//
588     double Etx = 0;//全体の送信消費電力
589     double Erx = 0;//全体の受信消費電力
590     double Etx1 = 0;//Fase1 送信消費電力
591     double Erx1 = 0;//Fase1 送信消費電力
592     double Etx2 = 0;//Fase2 送信消費電力
593     double Erx2 = 0;//Fase2 送信消費電力
594     double Etx3 = 0;//Fase3 送信消費電力
595     double Erx3 = 0;//Fase3 送信消費電力
596
597
598     int  kiseki[NODE];
```

```
599
600     for (i = 0; i < NODE; i++)
601     {
602         kiseki[i] = 0;
603     }
604     //=====================================================Fase１　全ノード収集フェーズ=============================================================//
605
606
607
608
609
610     for (i = 0; i < NODE; i++)
611     {
612         if (s[i] != 0)
613         {
614             now = i;
615             while (1)
616             {
617                 next = next_hop[now];
618                 Etx1 = Etx1 + Eelec*kdeta + omega*kdeta * distance2[now][next];
619                 Erx1 = Erx1 + Eelec*kdeta;
620                 kiseki[now] = 1;
621                 if (next == 0)
622                     break;
623                 now = next;
624             }
625         }
626     }
627
628     /*
629     for (i = 0; i < NODE; i++)
630     {
631         if (kiseki[i] == 1)
632         {
633             printf("%f¥t%f¥t%f¥t%d¥n", node[i].x, node[i].y, kansokuti[i], s[i]);
634         }
635     }
636
637     printf("%f¥t%f¥n", Etx1, Erx2);
638     */
639     //=====================================================Fase１　全ノード収集フェーズ完了=============================================================//
640     //=====================================================Fase2　送信ノード選別フェーズ=============================================================//
641     //全ノード送信時はコメントアウト
642     /*
643     for (i = 0; i < NODE; i++)
644     {
645         for (j = 0; j<NODE; j++)
646         {
647             if (s[i] == 1 && s[j] == 1 && i != j)
648             {
649                 if (distance2[i][j] <20)
650                 {
651                     s[j] = 0;
652                 }
653             }
654         }
655     }
656     for (i = 0; i < NODE; i++)
657     {
658         for (j = 0; j<NODE; j++)
```

```
659             {
660                 if (s[i] == 2 && s[j] == 2 && i != j)
661                 {
662                     if (distance2[i][j] <20)
663                     {
664                         s[j] = 0;
665                     }
666                 }
667             }
668         }
669         for (i = 0; i < NODE; i++)
670         {
671             for (j = 0; j<NODE; j++)
672             {
673                 if (s[i] == 3 && s[j] == 3 && i != j)
674                 {
675                     if (distance2[i][j] <20)
676                     {
677                         s[j] = 0;
678                     }
679                 }
680             }
681         }
682         for (i = 0; i < NODE; i++)
683         {
684             for (j = 0; j<NODE; j++)
685             {
686                 if (s[i] == 4 && s[j] == 4 && i != j)
687                 {
688                     if (distance2[i][j] <20)
689                     {
690                         s[j] = 0;
691                     }
692                 }
693             }
694         }
695         for (i = 0; i < NODE; i++)
696         {
697             for (j = 0; j<NODE; j++)
698             {
699                 if (s[i] == 5 && s[j] == 5 && i != j)
700                 {
701                     if (distance2[i][j] <20)
702                     {
703                         s[j] = 0;
704                     }
705                 }
706             }
707         }
708         for (i = 0; i < NODE; i++)
709         {
710             for (j = 0; j<NODE; j++)
711             {
712                 if (s[i] == 6 && s[j] == 6 && i != j)
713                 {
714                     if (distance2[i][j] <20)
715                     {
716                         s[j] = 0;
717                     }
718                 }
719             }
720         }
721
722
```

```
723
724
725
726  */
727
728
729
730
731
732
733
734
735
736
737      //=================================================Fase2　送信ノード選別フェーズ完了
           ==========================================================================//
738      //=================================================Fase3　データ収集フェーズ
           ==========================================================================//
739      int* layernumber = new int[NODE];
740      for (i = 0; i < NODE; i++)
741      {
742          layernumber[i] = 0;
743      }
744
745      for (i = 0; i < NODE; i++)
746      {
747          //if (s[i] != 0 && s[i] < 10)//提案方式時
748          if (s[i] == 1)//全ノード収集時
749          {
750              layernumber[i] = 1;
751              now = i;
752              while (1)
753              {
754                  next = next_hop[now];
755                  Etx2 = Etx2 + Eelec*kdeta + omega*kdeta * distance2[now][next];
756                  Erx2 = Erx2 + Eelec*kdeta;
757                  kiseki[now] = 2;
758                  if (next == 0)
759                      break;
760                  now = next;
761              }
762          }
763      }
764
765      for (i = 0; i < NODE; i++)
766      {
767          if (s[i] == 1)
768          {
769              //printf("%f¥t%f¥t%f¥n", node[i].x, node[i].y, kansokuti[i]);
770          }
771      }
772
773      for (i = 0; i < NODE; i++)
774      {
775          if (kiseki[i] == 2)
776          {
777              if (layernumber[i] == 1)
778              {
779                  //printf("%f¥t%f¥t%f¥t%d¥n", node[i].x, node[i].y, kansokuti[i], s
                      [i]);
780              }
781          }
782      }
```

```
783
784
785     //printf("%f¥t%f¥n", Etx1, Erx2);
786     //printf("%f¥t%f¥n", Etx2, Erx2);
787
788     //=================================================Fase2　データ収集フェーズ完了=================================================//
789
790
791     //=================================================補間フェーズ提案方式=================================================//
792
793     double* distance_to_sourcekuri = new double[EAREA];//各座標とイベント中心地との距離
794     double* eventnum = new double[EAREA]; //各座標に設定したイベント値
795     double** distancekuri = new double*[EAREA];//点とノードの距離
796     double** distancekuri2 = new double*[EAREA];//点とノードの距離の二乗
797     double*  distancesyuukei = new double[EAREA];//半径メートル以内にある送信観測ノードとの距離の合計
798     int*  kurinumber = new int[EAREA];//補間半径内に位置するノードの個数
799     double*  kansokutikuri1 = new double[EAREA]; //補間半径内のノードの観測値を距離で割った値を集計、計算用
800     double  hokanti[EAREA]; //kansokutikuri1/distancesyuukeiで割った値、補間によって設定された補間値
801
802     for (f = 1; f <= EAREA; f++)//初期化
803     {
804         distance_to_sourcekuri[f] = 0.0;
805         eventnum[f] = 0.0;
806         distancekuri[f] = new double[EAREA];
807         distancekuri2[f] = new double[EAREA];
808         distancesyuukei[f] = 0.0;
809         kansokutikuri1[f] = 0.0;
810         hokanti[f] = 0.0;
811         kurinumber[f] = 0;
812     }
813     for (i = 1; i <= EAREA; i++)//初期化
814     {
815         for (j = 0; j < NODE; j++)
816         {
817             distancekuri[i][j] = 0.0;
818             distancekuri2[i][j] = 0.0;
819         }
820     }
821
822     struct interpolation
823     {
824         double x;
825         double y;
826     } kuri[EAREA];
827
828     kuri[0].x = 0.0;
829     kuri[0].y = 0.0;
830     a = 0;
831     b = 0;
832     for (i = 1; i <= EAREA; i++)
833     {
834         kuri[i].x = a;
835         kuri[i].y = b;
836         x = kuri[i].x;
837         y = kuri[i].y;
838         b = b + 1;
839         if (b > EVENT)
840         {
```

```
841             a = a + 1;
842             b = 0;
843         }
844     }
845
846     for (i = 1; i <= EAREA; i++)//ノードiと発生源との距離
847     {
848         distance_to_sourcekuri[i] = sqrt((kuri[i].x - source_x)*(kuri[i].x -
              source_x) + (kuri[i].y - source_y)*(kuri[i].y - source_y));//三平方の定理
849         //printf("distance_to_source[%d]=%f¥n",i,distance_to_source[i]);//座標と発生
              源との距離の出力
850     }
851
852     for (i = 1; i <= EAREA; i++)
853     {
854         if (distance_to_sourcekuri[i] <= hankei)
855         {
856             eventnum[i] = (40 - distance_to_sourcekuri[i]) * (40 -
                  distance_to_sourcekuri[i]);
857             printf("%f¥t%f¥t%f¥n", kuri[i].x, kuri[i].y, eventnum[i]);
858         }
859     }//イベント半径内に位置する座標にイベント値を設定
860
861
862
863     /*
864     for (i = 1; i <= EAREA; i++)
865     {
866         if (distance_to_sourcekuri[i] <= hankei)
867         {
868             if (distance_to_sourcekuri[i] <= hankei2)
869             {
870                 eventnum[i] = tyuusinti;
871             }
872             if (distance_to_sourcekuri[i] > hankei2)
873             {
874
875             eventnum[i] = tyuusinti * 4 - (tyuusinti / 10) * distance_to_sourcekuri
                  [i];
876
877             }
878         }
879             //printf("%f¥t%f¥t%f¥n", kuri[i].x, kuri[i].y, eventnum[i]);
880
881     }
882     */
883
884     /*
885     for (i = 1; i <= EAREA; i++)
886     {
887         if (distance_to_sourcekuri[i] <= hankei)
888         {
889             eventnum[i] = tyuusinti - distance_to_sourcekuri[i] * ((tyuusinti - 10) /
                  hankei);
890             printf("%f¥t%f¥t%f¥n", kuri[i].x, kuri[i].y, eventnum[i]);
891         }
892     }//イベント半径内に位置する座標にイベント値を設定
893
894
895     */
896     /*
897     for (i = 0; i < NODE; i++)
898     {
899         if (distance_to_source[i] <= hankei)
```

```
900             {
901                 kansokuti[i] = (40 - distance_to_source[i]) * (40 - distance_to_source
                      [i]);
902             }
903         }
904         */
905
906         for (f = 1; f <= EAREA; f++)
907         {
908             for (i = 0; i < NODE; i++)
909             {
910                 //if (s[i] != 0 && s[i] <= 10)
911                 //if (s[i] != 0)
912                 if (s[i] != 0)
913                 {
914                     distancekuri[f][i] = sqrt((kuri[f].x - node[i].x)*(kuri[f].x - node
                          [i].x) + (kuri[f].y - node[i].y)*(kuri[f].y - node[i].y));//三平方
                          の定
915                     distancekuri2[f][i] = distancekuri[f][i] * distancekuri[f][i];
916                 }
917             }
918         }//収集されたノードとの距離を計算
919
920         for (f = 1; f <= EAREA; f++)
921         {
922             if (distance_to_sourcekuri[f] <= hankei)
923             {
924                 for (i = 0; i < NODE; i++)
925                 {
926                     //if (s[i] != 0 && s[i] < 10)
927                     //if (s[i] != 0)
928                     if (s[i] != 0)
929                     {
930                         if (kuri[f].x != node[i].x || kuri[f].y != node[i].y)
931                         {
932                             if (distancekuri[f][i] <= hokanhankei)
933                             {
934                                 distancesyuukei[f] = distancesyuukei[f] + (1 /
                                  distancekuri2[f][i]);
935                                 kurinumber[f]++;
936                             }
937                         }
938                     }
939                 }
940             }
941         }//補間半径内の収集ノードとの距離の逆数と個数を集計
942
943
944         for (f = 1; f <= EAREA; f++)
945         {
946             if (distance_to_sourcekuri[f] <= hankei)
947             {
948                 for (i = 0; i < NODE; i++)
949                 {
950                     //if (s[i] != 0 && s[i] < 10)
951                     //if (s[i] != 0)
952                     if (s[i] != 0)
953                     {
954                         if (kuri[f].x != node[i].x || kuri[f].y != node[i].y)
955                         {
956                             if (distancekuri[f][i] <= hokanhankei)
957                             {
958                                 kansokutikuri1[f] = kansokutikuri1[f] + (kansokuti[i] /
                                  distancekuri2[f][i]);
```

```
959                             }
960                         }
961
962                     }
963                 }
964             }
965             //kansokutikuri2[f] = kansokutikuri1[f] / distancesyuukei[f];
966         }//補間半径内のノードの観測値を距離で割り集計、その後その値を距離の逆数の集計値で割り、補間値を設定
967
968
969         for (f = 1; f <= EAREA; f++)
970         {
971             if (distance_to_sourcekuri[f] <= hankei)
972             {
973                 hokanti[f] = kansokutikuri1[f] / distancesyuukei[f];
974                 //printf("%f¥t%f¥t%f¥n", kuri[f].x, kuri[f].y, hokanti[f]);
975             }
976         }
977
978         for (f = 1; f <= EAREA; f++)
979         {
980             if (distance_to_sourcekuri[f] <= hankei)
981             {
982                 for (i = 0; i < NODE; i++)
983                 {
984                     //if (s[i] != 0 && s[i] < 10)
985                     //if (s[i] != 0)
986                     if (s[i] != 0)
987                     {
988                         if (kuri[f].x == node[i].x && kuri[f].y == node[i].y)
989
990                         {
991                             hokanti[f] = kansokuti[i];
992                         }
993                     }
994                 }
995             }
996         }//もし補間値を設定している座標が収集ノードの座標と一致している場合、そのノードの観測値をその座標の補間値として設定
997
998
999
1000
1001        for (i = 1; i <= EAREA; i++)
1002        {
1003            printf("%f¥t%f¥t%f¥n", kuri[i].x, kuri[i].y, eventnum[i]);
1004        }
1005
1006        for (i = 1; i <= EAREA; i++)
1007        {
1008                //printf("%f¥t%f¥t%f¥n", kuri[i].x, kuri[i].y, hokanti[i]);
1009        }
1010
1011        for (i = 1; i <= EAREA; i++)
1012        {
1013            if (distance_to_sourcekuri[i] <= hankei)
1014            {
1015                //printf("%f¥t%f¥t%f¥n", kuri[i].x, kuri[i].y, hokanti[i]);
1016            }
1017        }
1018
1019        for (i = 1; i < NODE; i++)
1020        {
```

```
1021            //if (s[i] != 0 && s[i] < 10)
1022            if (s[i] = 1)
1023            {
1024                //printf("%f¥t%f¥t%f¥n", node[i].x, node[i].y, kansokuti[i]);//収集したノードを表示
1025            }
1026        }
1027
1028        //==================================================補間フェーズ完了==========================================================//
1029
1030
1031    /*
1032    //==================================================補間フェーズ全ノード収集==========================================================//
1033
1034    double* distance_to_sourcekuri = new double[EAREA];//各座標とイベント中心地との距離
1035    double* eventnum = new double[EAREA]; //各座標に設定したイベント値
1036    double** distancekuri = new double*[EAREA];//点とノードの距離
1037    double** distancekuri2 = new double*[EAREA];//点とノードの距離の二乗
1038    double*  distancesyuukei = new double[EAREA];//半径メートル以内にある送信観測ノードとの距離の合計
1039    int*  kurinumber = new int[EAREA];//補間半径内に位置するノードの個数
1040    double*  kansokutikuri1 = new double[EAREA]; //補間半径内のノードの観測値を距離で割った値を集計、計算用
1041    double  hokanti[EAREA]; //kansokutikuri1/distancesyuukeiで割った値、補間によって設定された補間値
1042
1043    for (f = 1; f <= EAREA; f++)//初期化
1044    {
1045        distance_to_sourcekuri[f] = 0.0;
1046        eventnum[f] = 0.0;
1047        distancekuri[f] = new double[EAREA];
1048        distancekuri2[f] = new double[EAREA];
1049        distancesyuukei[f] = 0.0;
1050        kansokutikuri1[f] = 0.0;
1051        hokanti[f] = 0.0;
1052        kurinumber[f] = 0;
1053    }
1054    for (i = 1; i <= EAREA; i++)//初期化
1055    {
1056        for (j = 0; j < NODE; j++)
1057        {
1058            distancekuri[i][j] = 0.0;
1059            distancekuri2[i][j] = 0.0;
1060        }
1061    }
1062
1063    struct interpolation
1064    {
1065        double x;
1066        double y;
1067    } kuri[EAREA];
1068
1069    kuri[0].x = 0.0;
1070    kuri[0].y = 0.0;
1071    a = 0;
1072    b = 0;
1073    for (i = 1; i <= EAREA; i++)
1074    {
1075        kuri[i].x = a;
1076        kuri[i].y = b;
1077        x = kuri[i].x;
1078        y = kuri[i].y;
```

```
1079        b = b + 1;
1080        if (b > EVENT)
1081        {
1082            a = a + 1;
1083            b = 0;
1084        }
1085    }
1086
1087    for (i = 1; i <= EAREA; i++)//ノードiと発生源との距離
1088    {
1089        distance_to_sourcekuri[i] = sqrt((kuri[i].x - source_x)*(kuri[i].x - source_x) + (kuri[i].y - source_y)*(kuri[i].y - source_y));//三平方の定理
1090        //printf("distance_to_source[%d]=%f¥n",i,distance_to_source[i]);//座標と発生源との距離の出力
1091    }
1092
1093    for (i = 1; i <= EAREA; i++)
1094    {
1095        if (distance_to_sourcekuri[i] <= hankei)
1096        {
1097            eventnum[i] = tyuusinti - distance_to_sourcekuri[i] * ((tyuusinti - 10) / hankei);
1098            //printf("%f¥t%f¥t%f¥n", kuri[i].x, kuri[i].y, eventnum[i]);
1099        }
1100    }//イベント半径内に位置する座標にイベント値を設定
1101
1102
1103    for (f = 1; f <= EAREA; f++)
1104    {
1105        for (i = 0; i < NODE; i++)
1106        {
1107            if (distance_to_source[i] <= hankei)
1108            {
1109                distancekuri[f][i] = sqrt((kuri[f].x - node[i].x)*(kuri[f].x - node[i].x) + (kuri[f].y - node[i].y)*(kuri[f].y - node[i].y));//三平方の定
1110                distancekuri2[f][i] = distancekuri[f][i] * distancekuri[f][i];
1111            }
1112        }
1113    }//収集されたノードとの距離を計算
1114
1115    for (f = 1; f <= EAREA; f++)
1116    {
1117        if (distance_to_sourcekuri[f] <= hankei)
1118        {
1119            for (i = 0; i < NODE; i++)
1120            {
1121                if (distance_to_source[i] <= hankei)
1122                {
1123                    if (kuri[f].x != node[i].x || kuri[f].y != node[i].y)
1124                    {
1125                        if (distancekuri[f][i] <= hokanhankei)
1126                        {
1127                            distancesyuukei[f] = distancesyuukei[f] + (1 / distancekuri2[f][i]);
1128                            kurinumber[f]++;
1129                        }
1130                    }
1131                }
1132            }
1133        }
1134    }//補間半径内の収集ノードとの距離の逆数と個数を集計
1135
1136
1137    for (f = 1; f <= EAREA; f++)
```

```
1138  {
1139      if (distance_to_sourcekuri[f] <= hankei)
1140      {
1141          for (i = 0; i < NODE; i++)
1142          {
1143              if (distance_to_source[i] <= hankei)
1144              {
1145                  if (kuri[f].x != node[i].x || kuri[f].y != node[i].y)
1146                  {
1147                      if (distancekuri[f][i] <= hokanhankei)
1148                      {
1149                          kansokutikuri1[f] = kansokutikuri1[f] + (kansokuti[i] /
       distancekuri2[f][i]);
1150                      }
1151                  }
1152
1153              }
1154          }
1155      }
1156      //kansokutikuri2[f] = kansokutikuri1[f] / distancesyuukei[f];
1157  }//補間半径内のノードの観測値を距離で割り集計、その後その値を距離の逆数の集計値で割
       り、補間値を設定
1158
1159
1160  for (f = 1; f <= EAREA; f++)
1161  {
1162      if (distance_to_sourcekuri[f] <= hankei)
1163      {
1164          hokanti[f] = kansokutikuri1[f] / distancesyuukei[f];
1165          //printf("%f¥t%f¥t%f¥n", kuri[f].x, kuri[f].y, hokanti[f]);
1166      }
1167  }
1168
1169  for (f = 1; f <= EAREA; f++)
1170  {
1171      if (distance_to_sourcekuri[f] <= hankei)
1172      {
1173          for (i = 0; i < NODE; i++)
1174          {
1175              if (distance_to_source[i] <= hankei)
1176              {
1177                  if (kuri[f].x == node[i].x && kuri[f].y == node[i].y)
1178                  {
1179                      hokanti[f] = kansokuti[i];
1180                  }
1181              }
1182          }
1183      }
1184  }//もし補間値を設定している座標が収集ノードの座標と一致している場合、そのノードの観測
       値をその座標の補間値として設定
1185
1186
1187  for (i = 1; i <= EAREA; i++)
1188  {
1189      printf("%f¥t%f¥t%f¥n", kuri[i].x, kuri[i].y, hokanti[i]);
1190  }
1191
1192  for (i = 0; i < NODE; i++)
1193  {
1194      if (distance_to_source[i] <= hankei)
1195      {
1196          printf("%f¥t%f¥t%f¥n", node[i].x, node[i].y, kansokuti[i]);
1197      }
1198  }
```

```
1199
1200
1201  //=================================================補間フェーズ完了
        ==========================================================================//
1202  */
1203      //=================================================精度評価フェーズ
          ==========================================================================//
1204      double eventsyuukei = 0.0;//設定したイベント値の集計
1205      double gosa = 0.0;//誤差の集計
1206      double seido = 0.0;//最終的な精度,提案方式
1207
1208      for (f = 1; f <= EAREA; f++)
1209      {
1210          if (distance_to_sourcekuri[f] <= hankei)
1211          {
1212              if (eventnum[f] >= hokanti[f])//設定したイベント値の方が補間値よりも高い
                    場合
1213              {
1214                  eventsyuukei = eventsyuukei + eventnum[f];
1215                  gosa = gosa + eventnum[f] - hokanti[f];
1216              }
1217              if (eventnum[f] < hokanti[f])//設定したイベント値の方が補間値よりも低い場
                    合
1218              {
1219                  eventsyuukei = eventsyuukei + eventnum[f];
1220                  gosa = gosa + hokanti[f] - eventnum[f];
1221              }
1222          }
1223      }
1224
1225      seido = 1 - (gosa / eventsyuukei);
1226      printf("%f¥n", seido);
1227
1228      //=================================================精度評価フェーズ終了
          ==========================================================================//
1229
1230      /*delete[] distance;
1231      delete[] distance2;
1232      delete[] path;
1233      delete[] link;
1234      delete[] mark;
1235      delete[] distance_to_source;
1236
1237      delete[] s;
1238      delete[] edge;
1239      delete[] cell;
1240
1241      delete[] colle;
1242      delete[] colle_number;
1243      /*
1244      delete[] distance_to_sourcekuri;
1245      delete[] eventnum;
1246      delete[] distancekuri;
1247      delete[] distancekuri2;
1248      delete[] distancesyuukei;
1249      delete[] kurinumber;
1250      delete[] kansokutikuri1;
1251      delete[] hokanti;
1252      */
1253      return 0;
1254  }
1255
```