```
/*
【memo by 佐藤】

Outputファイル作成用に以下のディレクトリにフォルダを作成する
C:¥dataoutput¥WSN_hokan

地形を変えたい場合はベンチマーク関数を変えるとよい
ベンチマーク関数一覧↓
https://qiita.com/tomitomi3/items/d4318bf7afbc1c835dda

実行後，上記のフォルダに作成されるファイル
・hokan.dat ⇒ xy座標で表したmapのレイヤー方式の補間値
・hokan1d.dat ⇒ 上記のxy座標を1次元化したmapのレイヤー方式補間値
・hokanAll.dat ⇒ xy座標で表したmapの最短経路方式の補間値
・kansokuAllMap.dat ⇒ xy座標で表したmapの全イベント値
・kansokuAllMap1D.dat ⇒ 上記のxy座標を1次元化したmapの全イベント値
・kansokuMap.dat ⇒ xy座標で表したmapのランダムで配置したセンサノードが観測したイベント値
・nodeMap.dat ⇒ xy座標で表したmapのランダムで配置したセンサノードの座標
・NodeOfEachLayer.dat ⇒ 各レイヤーに属するセンサーノードの座標と観測値
・results.dat ⇒ 補間の精度と消費電力量
・Battery.dat ⇒ 各センサノードのバッテリ残量など

【記録】
・20180517
青木君のプログラム修正完了
色々とミスが見つかって修正した結果，補間精度がかなり向上した
必要な情報について，datファイルへの書き出しを実装した
レイヤー間隔と，レイヤーの中央値からの幅の値について，キーボード入力を実装した

・20180518
各センサノードのバッテリ残量を実装
将来的にシステム寿命を考慮した方向にする目的
補間の際に利用する近隣のノード数に条件を追加（補間半径内にkurinum以上ノードがない場合は補間半径を広げる処理）

・20180618
main文整理，関数などで外に出した

・20180704
ダイクストラ法修正

*/

#include <stdio.h>
#include <stdlib.h>
#define _USE_MATH_DEFINES
#include <math.h>
#include <time.h>
#include <iostream>
using namespace std;
#include <fstream>
#include <random>

#define EVENT 100//ベント領域の一辺
#define EAREA 10000//EVENT*EVENT
#define NODE 1000//配置するセンサの個数
#define RANGE 10//通信可能距離
#define RAND 1000//乱数調整
#define hokanhankei 10 //補間半径 = 通信可能距離
#define kurinum 2   //補間半径内にkurinum以上ノードがない場合は補間半径を広げる処理
#define batteryMax 100; //初期のバッテリー残量

//消費電力の各定数
```

```
 63  #define Eelec 0.0000000001
 64  #define kdeta 128//追加箇所
 65  #define omega 0.00000005//追加箇所
 66
 67
 68  //グローバル変数
 69  int x, y, a, b;
 70  int layernum = 0; //層の数
 71  int* s = new int[NODE]; //センサノードの状態（state）どのレイヤーに属するか
 72  double Battery1[NODE] = {0};    //全ノード収集フェーズでの各センサノードの消費電力量
 73  double Battery2[NODE] = {0};    //データ収集フェーズでの各センサノードの消費電力量
 74  double totalEtx1 = 0;   //Fase1 送信消費電力
 75  double totalErx1 = 0;   //Fase1 受信消費電力
 76  double totalEtx2 = 0;   //Fase2 送信消費電力
 77  double totalErx2 = 0;   //Fase2 受信消費電力
 78  double hokanhankei2 = hokanhankei;
 79  double hokanti[EAREA]; //kansokutikuri1/distancesyuukeiで割った値、補間によって設定さ
     れた補間値
 80  double hokanti2[EAREA];
 81  double* distance_to_sourcekuri = new double[EAREA];//各座標とイベント中心地との距離
 82  double* eventnum = new double[EAREA]; //各座標に設定したイベント値
 83  double** distancekuri = new double*[EAREA];//点とノードの距離
 84  double** distancekuri2 = new double*[EAREA];//点とノードの距離の二乗
 85  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
 86  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
 87  double*  kansokutikuri2 = 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
 88  double kansokutiAll[EVENT][EVENT]={};
 89  double kansokutiMax = 0.0;  //センサノードのイベント値の最大値
 90  double kansokutiMin = 10000.0;  //センサノードのイベント値の最小値
 91  double Battery[NODE] = {0};
 92  double kansokutiAl[EAREA] = {0};
 93  double layerdist = 0.0;
 94  double layerhaba = 0.0;
 95  int sleepNode[NODE] = {};
 96  int sleepNodex[NODE] = {};
 97  int sleepNodey[NODE] = {};
 98  int countB = 1; //消費電力の計算を何回繰り返したか
 99  int onehop[NODE] = {};
100  int twohop[NODE] = {};
101
102  int*  kurinumber = new int[EAREA];//補間半径内に位置するノードの個数
103
104  //各ノードの座標を格納
105  typedef struct zahyo{
106      int x;
107      int y;
108      int sleep = 0; // 1：ON,0：OFF
109      int sleepNodex[NODE];
110      int sleepNodey[NODE];
111  }Zahyo;
112  Zahyo node[NODE];
113  Zahyo sleepN[NODE];
114
115
116
117
118
119  typedef struct interpolation{
120      int x;
121      int y;
122  }Kuri;
```

```
123 Kuri kuri[EAREA];
124
125 void SensorNodeMap(int **mark){
126     node[0].x = 0;//SINKのx座標
127     node[0].y = 0;//SINKのy座標
128     mark[0][0] = 1;//SINKの位置は配置済み
129
130     for (int i = 0; i<RAND; i++){    //乱数調整 ノードのランダムな座標配置のために必要
131         rand();
132     }
133     /*コメント（佐藤）rand()関数は統計的にはあまり良い性質を持っていない、環境により
        実装が異なり再現性に難がある*/
134
135     for (int i = 1; i < NODE; i++)//i:ノード番号
136     {
137         while (1)//ノードiが配置し終わるまで続ける
138         {
139             node[i].x = rand() % EVENT;//((k%(EVENT/10))*10)～((k%(EVENT/10))*10)+9
                (0 - 32767をEVENTで割った余り) →EVENT=100の場合、確率的に0～67と68～
                99のので方が違う気がする（佐藤）
140             node[i].y = rand() % EVENT;//((k/(EVENT/10))*10)～((k/(EVENT/10))*10)+9
141             x = node[i].x;
142             y = node[i].y;
143             if (mark[x][y] != 0)//マーキングされていたら、配置しなおし
144             {
145                 continue;
146             }
147             mark[x][y] = 1;
148             break;
149         }
150
151     }
152
153     ofstream nodeMap;
154     nodeMap.open("C:/dataoutput/WSN_hokan/nodeMap.dat");
155     for (int i = 0; i < NODE ; i++) {
156         nodeMap << node[i].x << " " << node[i].y << endl;
157
158     }
159     cout << "¥n";
160
161     printf("=====センサノード配置MAP書き出し完了=====¥n");
162     printf("→ C:/dataoutput/WSN_hokan/nodeMap.dat¥n");
163     cout << "¥n";
164 }
165
166 void Link(double **distance, double **distance2, int **link){
167
168     for (int i = 0; i<NODE; i++){
169         //if (node[i].sleep == 1) continue;
170         for (int j = 0; j<NODE; j++){
171             if (node[j].sleep == 1) continue;
172             distance[i][j] = sqrt((double)(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
173             distance2[i][j] = distance[i][j] * distance[i][j];//ノード間距離の二乗の
                計算
174
175             if (distance[i][j] <= RANGE && i != j){
176                 link[i][j] = 1;
177             }else{
178                 link[i][j] = 0;
179                 //distance[i][j]=0;
180                 //distance2[i][j]=0;
```

```
181                 }
182             }
183         }
184     }
185
186     void OneHopfromSink(double **distance, int **link){
187         printf("SINKから1hopのノード¥n");
188         for(int i = 1; i < NODE; i++){
189             if(link[0][i]==1 && distance[0][i] <= RANGE){
190                 if(node[i].sleep == 1)continue;
191                 onehop[i] = 1;
192                 printf("(%d, %d) ",  node[i].x, node[i].y);
193             }
194         }
195         cout << "¥n¥n";
196
197     }
198
199     void TwoHopfromSink(double **distance, int **link){
200         printf("SINKから2hopのノード¥n");
201         for(int i = 0; i < NODE; i++){
202             if(onehop[i] == 1){
203                 for(int j = 0; j < NODE; j++){
204                     if(link[i][j]==1 && distance[i][j] <= RANGE && i != j){
205                         twohop[j] = 1;
206                         printf("(%d, %d) ",  node[j].x, node[j].y);
207                     }
208                 }
209             cout << "¥n";
210             }
211         }
212         cout << "¥n¥n";
213
214     }
215
216     void Dijkstra(double **path, int *use, int **link, double **distance, int *next_hop){
217         //参考URL:http://www.ss.cs.meiji.ac.jp/CCP024.html
218         //参考URL:http://www.deqnotes.net/acmicpc/dijkstra/
219
220         int max = 0;
221         int now = 0;
222         int next = 0;
223         int oldnext =0;
224         int count = 0;
225
226         for (int i = 0; i<NODE; i++){
227             for (int j = 0; j<NODE; j++){
228                 path[i][j] = 100000;
229             }
230             next_hop[i] = i;    //初期値は自分自身を指定
231             use[i] = 0;
232         }
233
234         now = 0;
235         path[0][0] = 0.0;   //シンクのコストを0に
236         for (int i = 1; i<NODE; i++)    //他のノードを無限大に
237         {
238             path[0][i] = 100000000;
239         }
240         use[0] = 1  ;   //シンクを使用済みに
241
242         //スリープノードを使用済みに
243         for(int i=0; i<NODE; i++){
244             if(sleepNode[i]==1){
```

```
245             use[i]=1;
246         }
247     }
248
249     while (1){  //全てのノードが使用済みになるまで繰り返す
250         max = 10000000;
251
252         for (int j = 0; j<NODE; j++){
253             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。
254                 path[0][j] = path[0][now] + distance[now][j];//最小コストの更新
255                 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
256             }
257             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
258                 next = j;
259                 max = (int)path[0][j];
260             }
261         }
262         if (now == next){   //次ノードがなかったら
263             for (int i = 1; i<NODE; i++){
264                 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
265                     if(next!=oldnext){
266                         now = count;
267                         oldnext = next;
268                     }else{
269                         while(count < NODE){
270                             count++;    //これ以上探索できないノードから次に進める（毎回0番に戻ると0番から進めなくなって他の未探索ノードが残ることがある）
271                             if(next_hop[count] != count){
272                                 now = count;
273                                 break;
274                             }
275                         }
276                         if(count >= NODE){
277                             goto LABEL3;
278                         }
279                     }
280                 }else{
281                     goto LABEL3;
282                 }
283             }
284         }else{
285             use[next] = 1;//次ノードを使用済みに
286             now = next;
287         }
288
289     }
290 LABEL3:
291     //for(int i=0; i<NODE;i++){
292     //  printf("%d ", use[i]);
293     //}
294
295     printf("=====全ノードからシンクまでの最短経路算出完了=====¥n");
296
297 }
298
299 void EventValueofAllnode(double kansokutiAllMin, double *kansokutiAl){
300     if(kansokutiAllMin < 0){
301         for (int i = 0; i < EVENT; i++){
302             for(int j = 0; j < EVENT; j++){
```

```
303                     kansokutiAll[i][j] -= kansokutiAllMin;
304                     kansokutiAl[j + i*EVENT] -= kansokutiAllMin;
305                 }
306             }
307         }
308 
309         ofstream kansokuAllMap;
310         kansokuAllMap.open("C:/dataoutput/WSN_hokan/kansokuAllMap.dat");
311         for (int i = 0; i < EVENT; i++){
312             for(int j = 0; j < EVENT; j++){
313                 kansokuAllMap << i << " " << j << " " << kansokutiAll[i][j] << endl;
314             }
315         }
316 
317         ofstream kansokuAllMap1D;
318         kansokuAllMap1D.open("C:/dataoutput/WSN_hokan/kansokuAllMap1D.dat");
319         for (int i = 0; i < EVENT*EVENT; i++){
320                 kansokuAllMap1D << i << " " << kansokutiAl[i] << endl;
321         }
322 
323         printf("=====全体の観測値MAP書き出し完了=====¥n");
324         printf("→ C:/dataoutput/WSN_hokan/kansokuAllMap.dat¥n");
325         cout << "¥n";
326 
327         printf("¥n=====イベント値設定完了 (Type：Eggholder function) =====¥n");
328 }
329 
330 void EventValueofSensorNode(double *kansokuti, int *use){
331 /*
332 ここではダイクストラ法によってSINKまでデータが届かないノードの情報は省く必要がある
333 */
334 
335         int nodeMax = 0;
336         int nodeMin = 0;
337 
338         for (int i = 0; i < NODE; i++){
339             for (int x = 0; x < EVENT; x++){
340                 for(int y = 0; y < EVENT; y++){
341                     if(node[i].x == x && node[i].y ==y){
342                         kansokuti[i] = kansokutiAll[x][y];
343                     }
344                 }
345             }
346         }
347 
348         for (int i = 0; i < NODE; i++){
349 
350             if(use[i]==1){      //SINKまで届かない観測値は外す
351                 //観測値の最大値
352                 if(kansokuti[i] > kansokutiMax){
353                     kansokutiMax = kansokuti[i];
354                     nodeMax = i;
355                 }
356                 //観測値の最小値
357                 if(kansokuti[i] < kansokutiMin){
358                     kansokutiMin = kansokuti[i];
359                     nodeMin = i;
360                 }
361             }
362         }
363 
364 
365         printf("¥n*****センサの観測値*****¥n");
366         printf("最大観測値 = %f (%d, %d), 最小観測値 = %f (%d, %d)¥n", kansokutiMax, node
```

```
          [nodeMax].x, node[nodeMax].y, kansokutiMin, node[nodeMin].x, node[nodeMin].y);
367       cout << "¥n";
368
369       //観測値マップ
370       ofstream kansokuMap;
371       kansokuMap.open("C:/dataoutput/WSN_hokan/kansokuMap.dat");
372       for (int i = 0; i < NODE ; i++) {
373           if(use[i]==1){
374               kansokuMap << node[i].x << " " << node[i].y << " " << kansokuti[i] <<
                    endl;
375           }
376       }
377
378       printf("=====配置されたセンサの観測値MAP書き出し完了=====¥n");
379       printf("→ C:/dataoutput/WSN_hokan/kansokuMap.dat¥n");
380       cout << "¥n";
381   }
382
383   void Layer(double *kansokuti){
384       double layer[100] ={}; //レイヤー（追加（佐藤））
385
386       cout << "観測値最小：" << kansokutiMin << ", 観測値最大：" << kansokutiMax <<
          endl; //キーボード入力追加（佐藤）
387       cout << endl;
388
389       if (countB == 1) {
390           cout << "【収集に使用する層のレイヤー数を入力してください】" << endl; //キー
              ボード入力追加（佐藤）
391           cin >> layernum;    // キーボードから入力を受ける
392           printf("¥n");
393       }
394
395       //layerhaba = tyuusinti / 20;//レイヤーの幅の決定（決め方が不明（佐藤））
396       layerdist = (kansokutiMax - kansokutiMin) / (layernum - 1);
397       layer[0] = kansokutiMin;
398
399       printf("レイヤーの間隔 = %f¥n", layerdist);
400
401       printf("¥n*****各レイヤーの中央値*****¥n");
402       for(int i=1; i<layernum ; i++){
403           layer[i] = layerdist * i + layer[0];
404       }
405       for(int i=0; i<layernum ; i++){
406           printf("layer[%d] = %f¥n", i, layer[i]);
407       }
408
409       if (countB == 1) {
410           printf("¥n");
411           cout << "【レイヤーの中央値からの幅を入力してください】" << endl; //キーボー
              ド入力追加（佐藤）
412           cout << "イベント領域大→幅を大きく，イベント領域小→幅を小さく" << endl;
413           cin >> layerhaba;    // キーボードから入力を受ける
414           printf("¥n");
415       }
416
417       //スリープモードのナンバー（Layernum）で初期化
418       for (int i = 0; i<NODE; i++){
419           s[i] = layernum;
420       }
421
422       //どのレイヤーに属するかを判定
423       for (int i = 0; i < NODE; i++){
424           for(int k= 0; k < layernum ; k++){
425               if (kansokuti[i] > layer[k] - layerhaba && kansokuti[i] < layer[k] +
```

```
                    layerhaba){
426                     s[i] = k;
427                 }
428             }
429         }
430
431         //各レイヤーのデータ送信ノード
432         printf("=====各レイヤーのデータ送信ノード書き出し完了=====¥n");
433         printf("→ C:/dataoutput/WSN_hokan/NodeOfEachLayer.dat¥n");
434         cout << endl;
435
436         ofstream NodeOfEachLayer;
437         NodeOfEachLayer.open("C:/dataoutput/WSN_hokan/NodeOfEachLayer.dat");
438         for (int k = 0; k < layernum; k++){
439             NodeOfEachLayer << "*****layer["<< k <<"]*****" << endl;
440             for (int i = 0; i < NODE; i++){
441                 if(s[i] == k){
442                     NodeOfEachLayer << node[i].x << " " << node[i].y << " " << kansokuti
                        [i] <<endl;
443                 }
444             }
445             NodeOfEachLayer << endl;
446         }
447
448         NodeOfEachLayer << "*****スリープモードのセンサーノード*****" << endl;
449         for (int i = 0; i < NODE; i++){
450             if(s[i] == layernum){
451                 NodeOfEachLayer << node[i].x << " " << node[i].y << " " << kansokuti[i]
                    <<endl;
452             }
453         }
454         NodeOfEachLayer << endl;
455
456     }
457
458     void BatteryFase1(int *next_hop, double **distance2, int **link){
459         double Etx1[NODE] = {0};//Fase1 送信消費電力
460         double Erx1[NODE] = {0};//Fase1 受信消費電力
461         int now = 0;
462         int next = 0;
463
464         totalEtx1 = 0;
465         totalErx1 = 0;
466
467         for (int i = 0; i < NODE; i++) {
468             Battery1[i] = 0;
469         }
470
471         for (int i = 1; i < NODE; i++){
472             //if(sleepNode[i] != 1){
473                 now = i;
474                 while (1){
475                     next = next_hop[now];
476                     if(next == now){
477                         break;
478                     }
479                     if(link[now][next] == 1){
480                         Etx1[now] = Etx1[now] + Eelec*kdeta + omega*kdeta * distance2
                            [now][next]; //送信電力
481                         Erx1[next] = Erx1[next] + Eelec*kdeta; //受信電力
482                     }
483
484                     now = next;
485
```

```
486                 if (next == 0){
487                     break;
488                 }
489 
490             }
491         //}
492     }
493 
494     for (int i = 0; i < NODE; i++){
495             totalEtx1 += Etx1[i];
496             totalErx1 += Erx1[i];
497 
498             Battery1[i] = Etx1[i] + Erx1[i];
499     }
500 
501     cout << "*****全ノード収集フェーズ 消費電力計算*****" << endl;
502     cout << "送信消費電力: " << totalEtx1 << ", 受信消費電力: " << totalErx1 << endl;
503     cout << "¥n";
504 }
505 
506 void BatteryFase3(int *next_hop, double **distance2, int **link){
507     double Etx2[NODE] = {0};//Fase2 送信消費電力
508     double Erx2[NODE] = {0};//Fase2 受信消費電力
509     int now = 0;
510     int next = 0;
511 
512     totalEtx2 = 0;
513     totalErx2 = 0;
514 
515     int* layernumber = new int[NODE];
516 
517     for (int i = 0; i < NODE; i++){
518         layernumber[i] = 0;
519         Battery2[i] = 0;
520     }
521 
522     for (int i = 0; i < NODE; i++){
523         //if (sleepNode[i]!=1){
524             if (s[i] < layernum) {
525                 layernumber[i] = 1;
526                 now = i;
527                 while (1) {
528                     next = next_hop[now];
529                     if(next == now){
530                         break;
531                     }
532                     if(link[now][next] == 1){
533                         Etx2[now] = Etx2[now] + Eelec*kdeta + omega*kdeta * distance2
                          [now][next];
534                         Erx2[next] = Erx2[next] + Eelec*kdeta;
535                     }
536 
537                     now = next;
538 
539                     if (next == 0){
540                         break;
541                     }
542 
543                 }
544             }
545         //}
546     }
547 
548     for (int i = 0; i < NODE; i++){
```

```
549             totalEtx2 += Etx2[i];
550             totalErx2 += Erx2[i];
551 
552             Battery2[i] = Etx2[i] + Erx2[i];
553     }
554 
555     cout << "*****選抜ノード収集フェーズ 1回分の消費電力計算*****" << endl;
556     cout << "送信消費電力: " << totalEtx2 << ", 受信消費電力: " << totalErx2 << endl;
557     cout << "¥n";
558 
559 }
560 
561 void Interpolation1(double *kansokuti){
562 
563     for (int f = 0; f < EAREA; f++)//初期化
564     {
565         distance_to_sourcekuri[f] = 0.0;
566         eventnum[f] = 0.0;
567         distancekuri[f] = new double[EAREA];
568         distancekuri2[f] = new double[EAREA];
569         distancesyuukei[f] = 0.0;
570         kansokutikuri1[f] = 0.0;
571         hokanti[f] = 0.0;
572         kurinumber[f] = 0;
573     }
574 
575     for (int i = 0; i < EAREA; i++)//初期化
576     {
577         for (int j = 0; j < NODE; j++)
578         {
579             distancekuri[i][j] = 0.0;
580             distancekuri2[i][j] = 0.0;
581         }
582     }
583 
584     kuri[0].x = 0;
585     kuri[0].y = 0;
586     a = 0;
587     b = 0;
588 
589     for (int i = 0; i < EAREA; i++){
590         kuri[i].x = a;
591         kuri[i].y = b;
592         x = kuri[i].x;
593         y = kuri[i].y;
594         b = b + 1;
595         if (b >= EVENT){
596             a = a + 1;
597             b = 0;
598         }
599     }
600 
601     int hokanhankei2 = hokanhankei;
602 
603     for (int f = 0; f < EAREA; f++){
604         for (int i = 0; i < NODE; i++){
605             if (s[i] < layernum){
606                 distancekuri[f][i] = sqrt((double)(kuri[f].x - node[i].x)*(kuri[f].x
                      - node[i].x) + (kuri[f].y - node[i].y)*(kuri[f].y - node[i].y));//
                      三平方の定理 d(s, si)
607                 distancekuri2[f][i] = distancekuri[f][i] * distancekuri[f][i];  //d
                      (s, si)^p (p = 2)
608             }
609         }
```

```
610     }//収集されたノードと全予測ノードの距離を計算
611
612     for (int f = 0; f < EAREA; f++){
613         for (int i = 0; i < NODE; i++){
614             if (s[i] < layernum){
615                 if (kuri[f].x != node[i].x || kuri[f].y != node[i].y){
616                     if (distancekuri[f][i] <= hokanhankei){
617                         distancesyuukei[f] = distancesyuukei[f] + (1 / distancekuri2[f][i]);    //sum(wj(s))
618                         kurinumber[f]++;
619                     }
620                 }
621             }
622         }
623         //補間半径内にkurinum以上ノードがなかった時に半径を増やす(佐藤)
624         while(kurinumber[f] <= kurinum){
625             kurinumber[f] = 0;
626             distancesyuukei[f] = 0;
627             hokanhankei2 += 2;
628             for (int i = 0; i < NODE; i++){
629                 if (s[i] < layernum){
630                     if (kuri[f].x != node[i].x || kuri[f].y != node[i].y){
631                         if (distancekuri[f][i] <= hokanhankei2){
632                             distancesyuukei[f] = distancesyuukei[f] + (1 / distancekuri2[f][i]);    //sum(wj(s))
633                             kurinumber[f]++;
634                         }
635                     }
636                 }
637             }
638         }
639         hokanhankei2 = hokanhankei;
640         //idw2 << "Σw" << f << "(S) :" << kuri[f].x << " " << kuri[f].y << " " << distancesyuukei[f] << " 補間に使うノード数 " << kurinumber[f] <<endl;
641         kurinumber[f] = 0;
642     }//補間半径内の収集ノードとの距離の逆数と個数を集計
643
644
645     //ofstream idw1;
646     //idw1.open("C:/dataoutput/WSN_hokan/idw1.dat");
647     for (int f = 0; f < EAREA; f++){
648         for (int i = 0; i < NODE; i++){
649             if (s[i] < layernum){
650                 if (kuri[f].x != node[i].x || kuri[f].y != node[i].y){
651                     if (distancekuri[f][i] <= hokanhankei){
652                         kansokutikuri1[f] = kansokutikuri1[f] + (kansokuti[i] / distancekuri2[f][i]);    //sum(wk(s)μ(sk)
653                         kurinumber[f]++;
654                     }
655                 }
656             }
657         }
658         //補間半径内にkurinum以上ノードがなかった時に半径を増やす(佐藤)
659         while(kurinumber[f] <= kurinum){
660             kurinumber[f] = 0;
661             kansokutikuri1[f] = 0;
662             hokanhankei2 += 2;
663             for (int i = 0; i < NODE; i++){
664                 if (s[i] < layernum){
665                     if (kuri[f].x != node[i].x || kuri[f].y != node[i].y){
666                         if (distancekuri[f][i] <= hokanhankei2){
667                             kansokutikuri1[f] = kansokutikuri1[f] + (kansokuti[i] / distancekuri2[f][i]);    //sum(wk(s)μ(sk)
668                             kurinumber[f]++;
```

```
669                             }
670                         }
671                     }
672                 }
673             }
674             hokanhankei2 = hokanhankei;
675             //idw1 << "Σw" << f << "(S)μ(S) : " << kuri[f].x << " " << kuri[f].y << " "
                  << kansokutikuri1[f] <<endl;
676         }//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
677
678
679         for (int f = 0; f < EAREA; f++){
680             for (int i = 0; i < NODE; i++){
681                 if (s[i] < layernum){
682                     if(kuri[f].x != node[i].x || kuri[f].y != node[i].y){
683                         hokanti[f] = kansokutikuri1[f] / distancesyuukei[f];
684                     }
685                 }
686             }
687         }
688
689         for (int f = 0; f < EAREA; f++){
690             for (int i = 0; i < NODE; i++){
691                 if (s[i] < layernum){
692                     if(kuri[f].x == node[i].x && kuri[f].y == node[i].y){
693                         hokanti[f] = kansokuti[i];
694                     }
695                 }
696             }
697         }//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
698
699
700         //for (int i = 1; i <= EAREA; i++){
701         //  printf("(%d, %d), 補間値：%f¥n", (int)kuri[i].x, (int)kuri[i].y, hokanti[i]);
702         //}
703
704         printf("=====補間値の書き出し完了=====¥n");
705         printf("→ C:/dataoutput/WSN_hokan/hokan.dat¥n");
706         cout << endl;
707
708         ofstream Hokan1D;
709         Hokan1D.open("C:/dataoutput/WSN_hokan/hokan1d.dat");
710         for (int i = 0; i < EAREA; i++){
711             Hokan1D << i << " " << hokanti[i] <<endl;
712         }
713
714         ofstream Hokan;
715         Hokan.open("C:/dataoutput/WSN_hokan/hokan.dat");
716         for (int i = 0; i < EAREA; i++){
717             Hokan << kuri[i].x << " " << kuri[i].y << " " << hokanti[i] <<endl;
718         }
719     }
720
721     void Interpolation2(double *kansokuti){
722         for (int f = 0; f < EAREA; f++)//初期化
723         {
724             hokanti2[f] = 0.0;
725             distance_to_sourcekuri[f] = 0.0;
726             eventnum[f] = 0.0;
727             distancesyuukei[f] = 0.0;
728             kansokutikuri1[f] = 0.0;
729             kurinumber[f] = 0;
```

```
730     }
731
732     for (int i = 0; i < EAREA; i++)//初期化
733     {
734         for (int j = 0; j < NODE; j++){
735             distancekuri[i][j] = 0.0;
736             distancekuri2[i][j] = 0.0;
737         }
738     }
739
740     for (int f = 0; f < EAREA; f++){
741         for (int i = 0; i < NODE; i++){
742                 distancekuri[f][i] = sqrt((double)(kuri[f].x - node[i].x)*(kuri[f].x
                    - node[i].x) + (kuri[f].y - node[i].y)*(kuri[f].y - node[i].y));//
                    三平方の定理 d(s, si)
743                 distancekuri2[f][i] = distancekuri[f][i] * distancekuri[f][i];  //d
                    (s, si)^p (p = 2)
744         }
745     }//全観測ノードと全予測ノードの距離を計算
746
747     hokanhankei2 = hokanhankei;
748
749     for (int f = 0; f < EAREA; f++){
750         for (int i = 0; i < NODE; i++){
751             if (kuri[f].x != node[i].x || kuri[f].y != node[i].y){
752                 if (distancekuri[f][i] <= hokanhankei){
753                     distancesyuukei[f] = distancesyuukei[f] + (1 / distancekuri2[f]
                        [i]);   //sum(wj(s))
754                     kurinumber[f]++;
755                 }
756             }
757         }
758         //補間半径内にkurinum以上ノードがなかった時に半径を増やす(佐藤)
759         while(kurinumber[f] <= kurinum){
760             kurinumber[f] = 0;
761             distancesyuukei[f] = 0;
762             hokanhankei2 += 2;
763             for (int i = 0; i < NODE; i++){
764                 if (kuri[f].x != node[i].x || kuri[f].y != node[i].y){
765                     if (distancekuri[f][i] <= hokanhankei2){
766                         distancesyuukei[f] = distancesyuukei[f] + (1 / distancekuri2
                          [f][i]);    //sum(wj(s))
767                         kurinumber[f]++;
768                     }
769                 }
770             }
771         }
772         hokanhankei2 = hokanhankei;
773         kurinumber[f] = 0;
774     }//補間半径内の収集ノードとの距離の逆数と個数を集計
775
776
777     for (int f = 0; f < EAREA; f++){
778         for (int i = 0; i < NODE; i++){
779             if (kuri[f].x != node[i].x || kuri[f].y != node[i].y){
780                 if (distancekuri[f][i] <= hokanhankei){
781                     kansokutikuri1[f] = kansokutikuri1[f] + (kansokuti[i] /
                        distancekuri2[f][i]);   //sum(wk(s)μ(sk)
782                     kurinumber[f]++;
783                 }
784             }
785         }
786         //補間半径内にkurinum以上ノードがなかった時に半径を増やす(佐藤)
787         while(kurinumber[f] <= kurinum){
```

```
788                 kurinumber[f] = 0;
789                 kansokutikuri1[f] = 0;
790                 hokanhankei2 += 2;
791                 for (int i = 0; i < NODE; i++){
792                     if (kuri[f].x != node[i].x || kuri[f].y != node[i].y){
793                         if (distancekuri[f][i] <= hokanhankei2){
794                             kansokutikuri1[f] = kansokutikuri1[f] + (kansokuti[i] /
                          distancekuri2[f][i]);    //sum(wk(s)μ(sk)
795                             kurinumber[f]++;
796                         }
797                     }
798                 }
799             }
800             hokanhankei2 = hokanhankei;
801         }//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
802
803
804         for (int f = 0; f < EAREA; f++){
805             for (int i = 0; i < NODE; i++){
806                 if(kuri[f].x != node[i].x || kuri[f].y != node[i].y){
807                     hokanti2[f] = kansokutikuri1[f] / distancesyuukei[f];
808                 }
809             }
810         }
811
812         for (int f = 0; f < EAREA; f++){
813             for (int i = 0; i < NODE; i++){
814                 if(kuri[f].x == node[i].x && kuri[f].y == node[i].y){
815                     hokanti2[f] = kansokuti[i];
816                 }
817             }
818         }//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
819
820
821         ofstream Hokan2;
822         Hokan2.open("C:/dataoutput/WSN_hokan/hokanAll.dat");
823         for (int i = 0; i < EAREA; i++){
824             Hokan2 << kuri[i].x << " " << kuri[i].y << " " << hokanti2[i] <<endl;
825         }
826     }
827
828     void AccuracyEvaluation(int countB){
829         double eventsyuukei = 0.0;  //設定したイベント値の集計
830         double gosa = 0.0;  //誤差の集計
831         double seido = 0.0; //最終的な精度,提案方式
832         double eventsyuukei2 = 0.0;
833         double gosa2 = 0.0;
834         double seido2 = 0.0;
835
836         //レイヤー方式
837         for (int f = 0; f < EAREA; f++){
838             if (kansokutiAl[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
839                 eventsyuukei = eventsyuukei + kansokutiAl[f];
840                 gosa = gosa + kansokutiAl[f] - hokanti[f];
841             }
842             if (kansokutiAl[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
843                 eventsyuukei = eventsyuukei + kansokutiAl[f];
844                 gosa = gosa + hokanti[f] - kansokutiAl[f];
845             }
846         }
```

```
847
848         seido = 1 - (gosa / eventsyuukei);
849         printf("%f¥n", seido);
850
851         //最短経路方式
852         for (int f = 0; f < EAREA; f++){
853             if (kansokutiAl[f] >= hokanti2[f]){      //設定したイベント値の方が補間値より
                も高い場合
854                 eventsyuukei2 = eventsyuukei2 + kansokutiAl[f];
855                 gosa2 = gosa2 + kansokutiAl[f] - hokanti2[f];
856             }
857             if (kansokutiAl[f] < hokanti2[f]){       //設定したイベント値の方が補間値より
                も低い場合
858                 eventsyuukei2 = eventsyuukei2 + kansokutiAl[f];
859                 gosa2 = gosa2 + hokanti2[f] - kansokutiAl[f];
860             }
861         }
862
863         seido2 = 1 - (gosa2 / eventsyuukei2);
864         printf("%f¥n", seido2);
865
866         //各センサのバッテリー残量の書き出し
867         ofstream battery;
868         battery.open("C:/dataoutput/WSN_hokan/Battery.dat");
869         battery << "データ送信回数 = " << countB << endl;
870         battery << endl;
871         for (int i = 0; i < NODE; i++){
872             battery << node[i].x << " " << node[i].y << " 全ノード収集フェーズ 消費電力計
                算： " << Battery1[i] << " 選抜ノード収集フェーズ 消費電力計算： " <<
                Battery2[i]*countB  << " バッテリー残量： " << Battery[i] << endl;
873         }
874         battery << endl;
875
876         printf("=====各センサノードのバッテリー残量の書き出し完了=====¥n");
877         printf("→ C:/dataoutput/WSN_hokan/Battery.dat¥n");
878         cout << endl;
879
880         ofstream Results;
881         Results.open("C:/dataoutput/WSN_hokan/results.dat");
882         Results << "レイヤー方式による逆距離荷重法の補間値精度：" << seido <<endl;
883         Results << "最短経路方式による逆距離荷重法の補間値精度：" << seido2 <<endl;
884         Results << endl;
885         Results << "*****全ノード収集フェーズ 消費電力計算*****" << endl;
886         Results << "送信消費電力: " << totalEtx1 << ", 受信消費電力: " << totalErx1 <<
              endl;
887         Results << endl;
888         Results << "*****選抜ノード収集フェーズ 消費電力計算*****" << endl;
889         Results << "送信消費電力: " << totalEtx2 << ", 受信消費電力: " << totalErx2 <<
              endl;
890         Results << "データ送信回数×送信消費電力: " << totalEtx2*countB << ", データ送信
              回数×受信消費電力: " << totalErx2*countB << endl;
891         Results << endl;
892         Results << "システムダウンまでのデータ送信回数 = " << countB << endl;
893         Results << endl;
894     }
895
896     //イベント値配置のための様々なベンチマーク関数
897     double FWP(int x, int y){
898         double xt=0, yt=0;
899
900         //関数の範囲によって変化する
901         xt = (double)40*x/(EVENT-1) - 20; // (-20 < x < 20)
902         yt = (double)40*y/(EVENT-1) - 20; // (-20 < y < 20)
903         return (500 * (1 - 1/(1+0.05*pow((xt*xt+yt-10), 2)) - 1/(1+0.05*((xt-10)*(xt-10)
```

```
          +yt*yt))
904         - 1.5/(1+0.03*((xt+10)*(xt+10)+yt*yt)) - 2/(1+0.05*((xt-5)*(xt-5)+(yt+10)*(yt
              +10)))
905         - 1/(1+0.1*((xt+5)*(xt+5)+(yt+10)*(yt+10))))*(1+0.0001*pow((xt*xt+yt*yt),
              1.2)));
906 }
907
908 double THC(int x, int y){
909     double xt=0, yt=0;
910
911     xt = (double)4*x/(EVENT-1) - 2; // (-2 < x < 2)
912     yt = (double)2*y/(EVENT-1) - 1; // (-1 < y < 1)
913     return 200 * ((4-2.1*xt*xt+pow(xt, 4)/3)*xt*xt + xt*yt + (-4+4*yt*yt)*yt*yt);
914 }
915
916 double gausu(int x,int y)
917 {
918     double xt = 0, yt = 0;
919     xt = (x - 50) / 20.0;
920     yt = (y - 50) / 20.0;
921     xt = 1 / sqrt(2 * M_PI)*exp(-(xt*xt) / 2.0);
922     yt = 1 / sqrt(2 * M_PI)*exp(-(yt*yt) / 2.0);
923     return  (EVENT-1)* xt * yt;
924 }
925 int main() {
926
927     printf("=====WSN観測値補間方式 START=====¥n");
928
929     //=====================================================変数定義＆初期化
          ====================================================//
930     int cont = 0;
931     int** link = new int*[NODE];
932     int use[NODE] = {};
933     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
934     int** mark = new int*[NODE];// 配置されているか判定に使う
935     int count = 0;
936     int* cell = new int[100];
937     int count0 = 1;
938     int number = 0;//イベント内のノードの個数
939     int edge_number = 0;//エッジノードの個数
940     int edge_number2 = 0;
941     int out_edge_number = 0;//未観測エッジノードの個数
942     int near = 0;
943     int head = 0;
944     int neighbor_node[NODE] = {};//隣接ノードの数
945     int neighbor_nonevent[NODE] = {};//隣接未収集ノードの数
946     int link_edge[NODE] = {};//通信可能範囲内のエッジノードの数
947     int* edge = new int[NODE];// (NODE)番ノードのエッジカウント
948     int** colle = new int*[NODE];
949     int* colle_number = new int[NODE];
950     int  saitan[NODE] = {};//追加箇所
951     int  ruisekideta[NODE] = {};
952     int  ruisekideta2 = 0;
953     int  cpu[NODE] = {};
954     int  cpu1[NODE] = {};
955     int sleepNodeLayerNumber[NODE] = {};
956
957     double d = 0;
958     double** distance = new double*[NODE];//ノード間距離
959     double** distance2 = new double*[NODE];//ノード間距離の二乗
960     double** path = new double*[NODE];
961     double source_x = 0.0;
962     double source_y = 0.0;
963     double kansokuti[NODE] = {};
```

```
964     double radius = 0.0;//イベントの半径
965     double batteryMax2[NODE];//更新用バッテリー
966     int SleepNode = 0; //スリープノードの番号を格納する
967     int SleepNode2 = 0; //更新用スリープノード番号
968
969     //srand((unsigned)time(NULL)); //ここを開くと乱数列が変わる
970
971     for (int 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
972     {
973         link[i] = new int[NODE];
974         mark[i] = new int[NODE];
975         distance[i] = new double[NODE];
976         distance2[i] = new double[NODE];
977         path[i] = new double[NODE];
978         colle[i] = new int[NODE];
979     }
980
981     for (int i = 0; i < NODE; i++)//初期化
982     {
983         node[i].sleep = 0;
984         for (int j = 0; j < NODE; j++)
985         {
986             distance[i][j] = 0.0;
987             distance2[i][j] = 0.0;
988             path[i][j] = 100000;
989             link[i][j] = 0;
990             mark[i][j] = 0;
991         }
992     }
993
994     //各センサノードのバッテリー量を定義（とりあえず100にしておいた）
995     for (int i = 0; i < NODE; i++) {
996         Battery[i] = batteryMax;
997     }
998     for (int i = 0; i < NODE; i++) {
999         batteryMax2[i]=100;
1000    }
1001    count = 0;
1002
1003
1004    //==================================================ランダム関数を使ったノード
          の座標配置==================================================//
1005
1006    SensorNodeMap(mark);
1007
1008    //==================================================ノード間距離とリンク
          ==================================================//
1009    while (1) {
1010        Link(distance, distance2, link);
1011
1012        //SINKからワンホップ以内に存在するセンサノード（1番消費電力が大きくなるノー
              ド）
1013        OneHopfromSink(distance, link);
1014
1015        //SINKからツーホップ以内に存在するセンサノード
1016        TwoHopfromSink(distance, link);
1017
1018        //==================================================ダイクストラ法による全
              てのセンサノードからの最短経路
              ==================================================//
1019        cout << "経路探索中" << "¥n";
1020        Dijkstra(path, use, link, distance, next_hop);
1021
1022        //==================================================イベントの発生とイベン
```

```
            ト観測値の設定==========================================================//
1023        if (countB == 1) {  //1回目のみ実行
1024
1025            double abs1 = 0.0;
1026            double abs2 = 0.0;
1027            double kansokutiAllMax = 0.0;    //全イベント値の最大値
1028            double kansokutiAllMin = 10000.0;    //全イベント値の最小値
1029            double kans = 0;
1030
1031            for (int i = 0; i < EVENT; i++) {
1032                for (int j = 0; j < EVENT; j++) {
1033
1034                    //ベンチマーク関数（ここをいろいろ変えてシミュレーションする）
1035                    //kans = FWP(i, j); // Five-well potential (FWP) function
1036                    //kans = THC(i,j); // Three-hump camel (THC) function
1037                    kans = gausu(i,j);
1038                    kansokutiAll[i][j] = kans;
1039                    kansokutiAl[j + i * EVENT] = kansokutiAll[i][j];
1040
1041                    //観測値の最大値
1042                    if (kansokutiAll[i][j] > kansokutiAllMax) {
1043                        kansokutiAllMax = kansokutiAll[i][j];
1044                    }
1045
1046                    //観測値の最小値
1047                    if (kansokutiAll[i][j] < kansokutiAllMin) {
1048                        kansokutiAllMin = kansokutiAll[i][j];
1049                    }
1050                }
1051            }
1052            cout << "¥n";
1053
1054            //すべてのノードのイベント値を設定
1055
1056            EventValueofAllnode(kansokutiAllMin, kansokutiAl);
1057
1058            //センサノードの観測値
1059            EventValueofSensorNode(kansokutiAll, use);
1060
1061            //==========================================================レイヤー方式の層の
                判定==========================================================//
1062
1063            Layer(kansokuti);
1064        }
1065
1066        //==========================================================消費電力の計算
            ==========================================================================//
1067
1068        //Fase１　全ノード収集フェーズの消費電力(SINKが全センサノードの情報を把握する
            ために一番最初に実行するフェーズでの消費電力)
1069        BatteryFase1(next_hop, distance2, link);
1070
1071        for (int i = 1; i < NODE; i++) {
1072            Battery[i] = Battery[i] - Battery1[i]; // i=0はSINKなのでバッテリーは減ら
                ない
1073        }
1074
1075        //Fase3　データ収集フェーズにおける消費電力(レイヤー方式で選別されたセンサ
            ノードがSINKにデータ送信する際の消費電力)
1076        BatteryFase3(next_hop, distance2, link);
1077
1078        //==========================================================システムダウンまで繰り
            返した時の各センサノードのバッテリ残量
```

```
                ================================================//
1079            //バッテリー残量で送信ノードをスリープモードに切り替えるときはここを修正
1080
1081            //データ収集フェーズ後のバッテリ残量
1082            double averageBattery = 0;
1083            double averageB = 0;
1084
1085
1086
1087            //どこかのバッテリー残量を50%間隔でスリープさせていく
1088            while (1) {
1089                for (int i = 1; i < NODE; i++) {
1090
1091                    Battery[i] = Battery[i] - Battery2[i];
1092                    SleepNode2 = SleepNode;
1093                    if (Battery[i] <= batteryMax2[i] *0.29) { //バッテリー残量が50%になっ
                          たバッテリーが出たら抜ける
1094                        printf("バッテリー50%のセンサ：(%d, %d), バッテリー残量：%f¥n",
                              node[i].x, node[i].y, Battery[i]);
1095                        printf("データ送信回数：%d¥n", countB);
1096                        node[SleepNode].x = sleepN[SleepNode].x;//スリープノードの番号を
                              更新する（2週目から）
1097                        node[SleepNode].y = sleepN[SleepNode].y;
1098                        sleepN[i].x = node[i].x; //スリープノードの座標をsleepNに格納
1099                        sleepN[i].y = node[i].y;
1100                        printf("sleepnode：(%d, %d), バッテリー残量：%f¥n", sleepN[i].x,
                              sleepN[i].y, Battery[i]);
1101                        node[i].x = 1000;//ノードの座標を移動
1102                        node[i].y = 1000;
1103                        batteryMax2[i] = Battery[i]; //batteryMax2の値を更新
1104                        SleepNode = i; //スリープノードの番号を格納
1105                        printf("ノード番号：%d¥n", SleepNode);
1106                        break;
1107                    }
1108                }
1109                countB++;
1110
1111
1112                if (SleepNode != SleepNode2) {
1113                    break;
1114                }
1115            }
1116            LABEL:
1117            if (batteryMax2[SleepNode] < 0){
1118                break;
1119            }
1120        }
1121
1122
1123        cout <<endl;
1124
1125        //=================================================補間フェーズ
              ================================================//
1126
1127        cout << "=====観測値補間フェーズ（逆距離荷重法）=====" << endl;
1128        cout <<endl;
1129
1130        //補間フェーズレイヤー方式
1131        Interpolation1(kansokuti);
1132
1133        //補間フェーズ全センサノード収集方式
1134        Interpolation2(kansokuti);
1135
1136        //=================================================精度評価フェーズ
```

```
        ================================================//
1137
1138        AccuracyEvaluation(countB);
1139
1140        return 0;
1141    }
1142
1143
```