Functions	
d3batch	1:46

```
/* delaunay3 Delaunay/Power diagrams in 3d Jack Snoeyink Aug 2003
 Implements Watson's incremental Delaunay, with sphere-based search and
 simplical data structure storing vertices and opposite neighbors.
 Points must be scaled to integers; guaranteed for differences of 10
 bits, and uses leveling and hilbert curve to guarantee 16 bits if the
 points are well distributed.  Works for pdb files, which are 20 bits.
 Handles degeneracies by perturbing points by increasing infinitesimals
 to guarantee that all simplices are full-dimensional.
 */

#ifndef DELAUNAY3_H
#define DELAUNAY3_H

#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef float coord; // We'll just use floats for x,y,z coordinates

typedef struct { // Basic point data structure element
  int index; // index back to original line in file.
  coord x,y,z; double sq;   // coordinates: 3 spatial + lifted
} pointType, *ppointType;

//tetrahedra are groups of four consecutive corners
typedef struct cornerType { // data associated with each corner
  ppointType v; // index of vertex
  int opp; // pointer to opposite corner in neighboring tetra
} cornerType, *pcornerType;

/* Tetrahedra are groups of four corners in order of increasing
 vertex index, except that the first two may be swapped to ensure
 that the orientation determinant is positive.
 I.e., take the lex smallest alternating permutation with positive sign.
 There must always be an odd number of swaps between two permutations;
 we swap the first two if necessary to achieve this.  */

/* d3.c Delaunay/Power diagram function
 d3batch takes input vertices, and returns a (compact) corner table for Delaunay.
 REQUIRES that the first point is at infinity,
 and that the first 5 are in general position.
 (I should probably verify or relax this.)
 Since all points have radii assigned already, it can handle power diagrams.
 */
void d3batch(ppointType vert, int nvert, // input vertices (pointType[]) & number of vertices
        pcornerType *result, int *ncorners); // output corner table


#define XX 0
#define YY 1
#define ZZ 2

#define NVERT      5000 // number of random vertices
#define MAXVERT 200000 // pdb has 5 digit atom# field
#define TETperV    10 //
#define FORGETLIMIT  10000 // forget tetra after this many (for better locality of ref? Doesn't help.)

// Some useful definitions
#define EQUALV(i,j) (vert[i].x == vert[j].x && vert[i].y == vert[j].y && vert[i].z == vert[j].z)
#define EQUALPV(pv,v) (pv->x == v->x && pv->y == v->y && pv->z == v->z)

#define DET2(p,q,i,j)((i##p)*(j##q) - (j##p)*(i##q))
#define DOT(a,b) ((a)->x*(b)->x + (a)->y*(b)->y + (a)->z*(b)->z)

// Stack data structure operations
#define STACKMAX 2000
#define POP(stack) (stack##st[stack##sp--])
#define isEMPTY(stack) (stack##sp < 0)
#define stkINIT(stack) {stack##sp = -1; }

#ifndef STATS
#define PUSH(value, stack) { stack##st[++stack##sp] = value; }
#define stkDECLARE(stack,stn) int stack##sp, stack##st[STACKMAX];
#else
#define PUSH(value, stack) { \
    stack##st[++stack##sp] = value; \
    if (stack##max < stack##sp) { stack##max = stack##sp; \
      if (stack##max >= STACKMAX) { \
        printf("ERROR: overflow stack %x pushing %d",  stack##st, value); exit(EXIT_FAILURE); } } /**/\
    }
#define stkDECLARE(stack,stn) int stack##sp, stack##st[STACKMAX]; int stack##max; //AUDIT /**/
#endif

typedef struct sphereType { // sphere equation
  double x, y, z, sq; // Invariant sq > 0 for all created tetra, unless they use pt at infty
 } sphereType, *psphereType;

typedef struct {
  ppointType vert; // vertices: 0th is point at infinity!!!!
  pcornerType s;   // corner table
  psphereType sph; // spheres
  int *active; // which spheres are active
  int freeTetra; // head for free list for tetrahedra kept in opp[CORNER(tetra,0)]
  int liveTetra; // latest tetra; known to be live.
```

```
96      int maxTetra; // AUDIT only
97      int limitmaxTetra; // limit on # of created tetrahedra, spheres & corners/4.
98      // stacks used in inserting pv
99      stkDECLARE(dfs, "dfs");   // DFS stack to find dead tetras
100     stkDECLARE(idfs, "idfs"); // DFS stack for tetras adj to infinite vertex (>4*30)
101     stkDECLARE(nhbr, "nhbr"); // stack for dead corners with live neighbors
102     stkDECLARE(kill, "kill"); // stack for base corners of tetras to recycle
103   } d3stateType, *pd3stateType;
104
105   // readers call setVert when they make a point
106   void setVert(ppointType v, int indx, double xx, double yy, double zz,
107                double rad, double mult);
108
109   #define VSUBASSN(vin, xx,yy,zz, vout) {\
110       vout->index = vin->index;  vout->sq = vin->sq;\
111       vout->x = vin->x - xx; vout->y = vin->y - yy; vout->z = vin->z - zz;\
112     }
113
114   //Some compiler/unix variants
115   #ifndef FALSE
116   #define FALSE 0
117   #endif
118
119   #ifndef TRUE
120   #define TRUE 1
121   #endif
122
123   #ifndef strcasecmp
124   #define strcasecmp(s1,s2)       strcmp(s1,s2)
125   #define strncasecmp(s1,s2,n)    strncmp(s1,s2,n)
126   #endif
127
128   #define HIGHCOORD 0x4000
129   #define COORDMASK 0x3fff
130
131   #ifdef bcc
132   #define RANDBIT (random(2))                 // one random bit
133   #define RAND2BIT (random(4))                // two random bits
134   #define RANDPROB(mask) (random(mask+1))     // random bits masked
135   #define RANDCOORD (double)random(HIGHCOORD);
136   #define RANDOM(k) random(k)                 // random number 0..k-1
137   #else
138   #define RANDBIT (random()&1)                // one random bit
139   #define RAND2BIT (random()&3)               // two random bits
140   #define RANDPROB(mask) (mask&random())      // random bits masked
141   #define RANDCOORD (double)(random()&COORDMASK)
142   #define RANDOM(k) random()%(k)              // random number 0..k-1
143   #endif
144
145   //  fprintf(stderr, "\nASSERT FAILED (line %d of %s ): %s\n",
146   #ifndef NOASSERT
147   #define ASSERT(bool, string) if (!(bool)) {\
148     printf("\nASSERT FAILED (line %d of %s ): %s\n", \
149             __LINE__, __FILE__, string); }
150   #else
151   #define ASSERT(bool, string)
152   #endif
153
154   #endif
155
```

Functions

Functions	
d3batch	1:16

```
1   /* d3 Delaunay triangulator in 3d          Jack Snoeyink Aug 2003
2    Implements Watson's incremental Delaunay, with sphere-based search and
3    simplical data structure storing vertices and opposite neighbors.
4    Handles degeneracies by perturbing points by increasing infinitesimals.
5    Guaranteed for integer coordinates of 10 bits. No flat simplices.
6    */
7
8   #include <math.h>
9
10  /* d3.c Delaunay/Power diagram function
11   d3batch takes input vertices, and returns a (compact) corner table for Delaunay.
12   REQUIRES that the first point is at infinity,
13   and that the first 5 are in general position. (I should verify or relax this.)
14   Since all points have radii assigned already, it can produce power diagrams.
15   */
16  void d3batch(ppointType vert, int nvert, // input vertices (pointType[]) & number of vertices
17          pcornerType *result, int *ncorners); // output corner table
18
19  /* void d3initialize(const pd3stateType this, ppointType vertArray, int nvert)
20   initialize d3state this from vertArray.  Allocates memory for corners and spheres,
21   sets up the free list for tetrahedra, and creates spheres for the first five points.
22   REQUIRES: vertArray[0] contains the point at infinity and vertArray[1..4] contain
23   four finite points; these first five points must be in general position.
24   */
25  void d3initialize(const pd3stateType this, ppointType vertArray, int nvert);
26
27  // LOCATION ROUTINES walk a mesh stored in s, active, & sph starting from corner start to find a point pv.
28  // result is the index of a simplex whose sphere strictly contains pv.
29  // (The simplex itself need not contain pv.)
30  // The return code is positive if we succeed (# of location steps at present)
31  // Return code of 0 means that we failed, perhaps because points with small weight have no Voronoi cells
32  // Return code of -1 means that we found the duplicate of a vertex in the mesh. IN THIS CASE, result
33  //   is the location of the corner where we found the duplicate!
34  int d3locSphere(const pd3stateType this, ppointType pv, int start,
35                      int *result); // output
36
37  /* d3insert(const pd3stateType this, int vi, int p)
38   inserts the point this->vert[vi] that is contained in sphere p
39   into the delaunay triangulation stored in this.
40   (d3locSphere may be used to obtain p.)
41   */
42  void d3insert(const pd3stateType this, int vi, int p);
43
44  /* d3compactCorners(const pd3stateType this, pcornerType *result, int *ncorners);
45   Takes corner table this->s in which some corners/tetrahedra are unused, and
46   returns a compactified corner table result, and its length ncorners.
47   DESTROYS the corner table s and active flags in the process.
48   */
49  int d3compactCorners(const pd3stateType this, // arrays this->s & this->active are DESTROYED!
50                      pcornerType *result, int *ncorners); // output
51
52  // Functions to access corner tables
53  #define MOD4(a) (a & 3)
54  #define TETRA(corner) ((corner) >> 2)
55  #define INDEX(corner) (MOD4(corner))
56  #define CORNER(tetra,index) (((tetra)<<2)+(index))
57  #define BASECORNER(corner) ((corner) & 0xFFFFFFFC)
58  #define LASTCORNER(corner) ((corner)|3)
59
60  #define DEAD(p) (this->active[p] <= 0) // is this a dead or killed tetrahedron?
61  #define KILL(p) {this->active[p] = -1;} // kill tetrahedron
62  //#define DEAD(p) (sph[p].sq < 0) // is this a dead tetrahedron?
63  //#define KILL(p) {sph[p].sq = -1;} // kill tetrahedron
64
65  #define infiniteV(pv,vert) ((pv) == vert) // first point is at infinity
66  #define infiniteC(c) infiniteV(this->s[c].v, this->vert) // corner uses inf pt
67  #define infiniteP(p) (this->sph[p].sq == 0.0) // if tetra uses infinite point
68
69  /* Tetrahedra are groups of four corners in order of increasing
70   vertex index, except that the first two may be swapped to ensure
71   that the orientation determinant is positive.
72   I.e., take the lex smallest alternating permutation with positive sign.
73   There must always be an odd number of swaps between two permutations;
74   we swap the first two if necessary to achieve this.
75   */
76
77  // set corner's vertex and opposite in tetrahedron structure
78  //void setCornerVC(int c, int vv, int op) {
79  #define setCornerVC(c, vv, op) { this->s[c].v = vv; this->s[c].opp = op;}
80  #define setCornerPairV(c, op, vv, ov) { setCornerVC(c, vv, op); setCornerVC(op, ov, c); }
81  #define setCornerVCN(c, vv, op) { setCornerVC(c, vv, op); this->s[op].opp = c; } // set corner & adjust nhbr
     opp
82
```

Functions


```
/* d3 Delaunay triangulation in 3d          Jack Snoeyink Aug 2003
 Implements Watson's incremental Delaunay, with sphere-based location,
 and simplical data structure storing vertices and opposite neighbors.
 Handles degeneracies by perturbing points by increasing infinitesimals.
 Guaranteed for integer coordinates of 10 bits. No flat simplices.
 */

#include "delaunay3.h"
#include "d3.h"


#define spdot(sp,pv,sv) ((sp)->x*((pv)->x-(sv)->x)+(sp)->y*((pv)->y-(sv)->y)\
         +(sp)->z*((pv)->z-(sv)->z)+(sp)->sq*((pv)->sq-(sv)->sq))
#define spdotInf(sp,pv) (sp)->sq

/* Allocate or free space for a tetrahedron.
 When allocating, this->liveTetra is the location fo the new tetrahedron.
 */
#define STARTTETRA(this) \
{ startTetraCnt++; (this)->liveTetra = (this)->freeTetra; \
  (this)->freeTetra = (this)->s[CORNER((this)->liveTetra,0)].opp;  \
  ASSERT(DEAD((this)->liveTetra), "Reusing existing tetrahedron?"); (this)->active[(this)->liveTetra] = 1; \

  if ((this)->maxTetra <= (this)->liveTetra) { (this)->maxTetra = (this)->liveTetra+1; \
  if ((this)->liveTetra >= (this)->limitmaxTetra) { printf("AUDIT: %d > limitmaxTetra\n", (this)->liveTetra)
; exit(EXIT_FAILURE); }}/**/\
} // AUDIT

#define FREETETRA(this, p) \
{ freeTetraCnt++;  /*TESTING*/ ASSERT((this)->active[p]<0, "Freeing already free tetrahedron?"); /**/\
  (this)->active[p]=0;  if (((this)->maxTetra-p)<FORGETLIMIT) {\
  (this)->s[CORNER(p,0)].opp = (this)->freeTetra; (this)->freeTetra = p; /**/  \
}}

//Tetrahedron manipulation tables
static const short offset[4][4] = { // c+offset[i][INDEX(c)] advances c to (c+i)mod5
  /*0*/{0,0,0,0}, /*1*/{1,1,1,-3}, /*2*/{2,2,-2,-2}, /*3*/{3,-1,-1,-1}};
#define INCREMENT(c) (c+offset[1][INDEX(c)])
// drop[i] contains new vertex order after vertex i is dropped and replaced by pv on same side.
// offdr[i] contains drop(i)-index(i)
// invdrop[i][k] = j whenever drop[i][j] = k.  4s signal i=k; bad because i is dropped.
static const short drop[4][3] = {{2,1,3}, {0,2,3}, {1,0,3}, {0,1,2}};
static const short offdr[4][3] = {{2,1,3}, {-1,1,2}, {-1,-2,1}, {-3,-2,-1}};
static const short invdrop[4][4] = {{4,1,0,2}, {0,4,1,2}, {1,0,4,2}, {0,1,2,4}};

#include "d3audit.c"

inline double InSpherev(psphereType sp, ppointType pv, ppointType sv) {
  double d = spdot(sp, pv, sv); // Return true if inside (==negative)
  inSphereCnt++; /*TESTING*/
  return d; // perturb those on sphere to inside
}

inline void makeSphereV(psphereType sp, ppointType v0, ppointType v1, ppointType v2, ppointType pv, ppointTy
pe vert) {
  double x0, y0, z0, sq0, x1, y1, z1, sq1, x2, y2, z2, sq2;
  double xy, xz, xs, yz, ys, zs; // 2x2 minors
  // make sphere: only v0 or v1 may be infinte.
  sphereCnt++; /*TESTING*/
  if(!infiniteV(v0, vert)) {
    x0 = v0->x - pv->x; y0 = v0->y - pv->y;
    z0 = v0->z - pv->z; sq0 = v0->sq - pv->sq;
  } else { x0 = v0->x; y0 = v0->y; z0 = v0->z; sq0 = v0->sq; }
  if(!infiniteV(v1, vert)) {
    x1 = v1->x - pv->x; y1 = v1->y - pv->y;
    z1 = v1->z - pv->z; sq1 = v1->sq - pv->sq;
  } else { x1 = v1->x; y1 = v1->y; z1 = v1->z; sq1 = v1->sq; }
  x2 = v2->x - pv->x; y2 = v2->y - pv->y;
  z2 = v2->z - pv->z; sq2 = v2->sq - pv->sq;
  xy = DET2(0,1,x,y);
  xz = DET2(0,1,x,z);
  yz = DET2(0,1,y,z);
  xs = DET2(0,1,x,sq);
  ys = DET2(0,1,y,sq);
  zs = DET2(0,1,z,sq);
  sp->x  = -y2*zs +z2*ys -sq2*yz;
  sp->y  =  x2*zs -z2*xs +sq2*xz;
  sp->z  = -x2*ys +y2*xs -sq2*xy;
  sp->sq =  x2*yz -y2*xz +z2*xy;
  // sp->w = -p2->x*sp->x -p2->y*sp->y -p2->z*sp->z -p2->sq*sp->sq;
  /* printf("disp('Sphere equation: <%5.0f %5.0f %5.0f %5.0f %5.0f>')\n",sp->x,sp->y,sp->z,sp->sq);
   (void)fflush(stdout);
   printf("%%Sp ck: %g %g %g %g\n", spdot(sp,v0), spdot(sp,v1), spdot(sp,v2), spdot(sp,pv)); /**/
}
```

Functions	
InSpherev	1:46
d3initialize	2:104
makeSphereV	1:52

```
84
85    // when we initialize, this is what we fill in.
86    //const int initialopp[] = {11,5,15,21,25, 1,10,16,20,26, 6,0,17,22,27, 2,7,12,23,28, 8,3,13,18,29, 4,9,14,1
      9,24};
87    static const int initialopp[5][4] = {
88      {CORNER(1,1), CORNER(2,0), CORNER(3,1), CORNER(4,0)},
89      {CORNER(2,1), CORNER(0,0), CORNER(3,0), CORNER(4,1)},
90      {CORNER(0,1), CORNER(1,0), CORNER(3,2), CORNER(4,2)},
91      {CORNER(1,2), CORNER(0,2), CORNER(2,2), CORNER(4,3)},
92      {CORNER(0,3), CORNER(1,3), CORNER(2,3), CORNER(3,3)}};
93
94    static const int initialv[2][5][4] = {{{1,2,3,4}, {2,0,3,4}, {0,1,3,4}, {1,0,2,4}, {0,1,2,3}},
95                                          {{0,2,3,4}, {2,1,3,4}, {1,0,3,4}, {0,1,2,4}, {1,0,2,3}}};
96
97
98    /* void d3initialize(const pd3stateType this, ppointType vertArray, int nvert)
99     initialize d3state this from vertArray.  Allocates memory for corners and spheres,
100    sets up the free list for tetrahedra, and creates spheres for the first five points.
101    REQUIRES: vertArray[0] contains the point at infinity and vertArray[1..4] contain
102    four finite points; these first five points must be in general position.
103    */
104   void d3initialize(const pd3stateType this, ppointType vertArray, int nvert) {
105     int j, p, last;
106     double d;
107
108     //  initBitTable(); // BITS
109     this->vert = vertArray;
110     this->limitmaxTetra = TETperV*nvert; // allocate space for spheres and corners
111     this->s   = (pcornerType) calloc(4*this->limitmaxTetra, sizeof(cornerType)); // per corner: v, opp
112     this->sph = (psphereType) calloc(this->limitmaxTetra, sizeof(sphereType)); // per tetra: sphere eqn
113     this->active = (int *) calloc(this->limitmaxTetra, sizeof(int)); // flag -1 unused, 0 dead, 1 alive
114
115     if ((this->s == NULL) || (this->sph == NULL) || (this->active == NULL)) {
116       printf("ERROR: d3batch could not calloc memory for data structures\n");
117       exit(EXIT_FAILURE);
118     }
119
120     // initialize tetrahedra
121     last = -1; /* set up free list of tetrahedra */
122     this->freeTetra = this->limitmaxTetra;
123     do {
124       this->freeTetra--;
125       this->active[this->freeTetra] = 0; // KILL(freeTetra);
126       this->s[CORNER(this->freeTetra,0)].opp = last;
127       last = this->freeTetra;
128     } while (this->freeTetra > 5);
129
130     this->active[4] = 2; // create first sphere
131     makeSphereV(this->sph+4, this->vert+0, this->vert+1, this->vert+2, this->vert+3, this->vert);
132     d = spdot(this->sph+4, this->vert+4, this->vert+3); // if d<0, then we need to swap
133
134     if (d == 0.0) {
135       printf("ERROR: Need first five vertices to be in general position"); exit(EXIT_FAILURE);
136     }
137
138     if (d < 0) {
139       this->sph[4].x = -this->sph[4].x; this->sph[4].y = -this->sph[4].y; this->sph[4].z = -this->sph[4].z;
140       this->sph[4].sq = -this->sph[4].sq;
141     }
142
143     for (p=0; p<5; p++) {
144       for (j=0; j<4; j++) {        // pay attention to orientation when assigning vertices
145         setCornerVC(CORNER(p,j), this->vert+initialv[d<0][p][j], initialopp[p][j]);
146       }
147       makeSphereV(this->sph+p, this->s[CORNER(p,0)].v, this->s[CORNER(p,1)].v, this->s[CORNER(p,2)].v, this->s
      [CORNER(p,3)].v, this->vert);
148       this->active[p] = 1;
149       // swap first two if d<0.
150       if (((d<0)&&(p==1)) || ((d>0) && (p==0)))
151         {ASSERT(this->sph[p].sq >0, "Somehow vertp at infinity is in or on sphere p in init");}
152       else
153         {ASSERT(spdot(this->sph+p, this->vert+p + (d<0)*(p<2)*(1-2*p), this->s[CORNER(p,3)].v) > 0,
154                 "Somehow vertp is in or on sphere p in init.");}
155     }
156     this->liveTetra = 4;
157     this->maxTetra = 5;
158   }
159
160
161
```

Functions


```
162
163 // LOCATION ROUTINES walk a mesh stored in s, active, & sph starting from corner start to find a point pv.
164 // result is the index of a simplex whose sphere strictly contains pv.
165 // (The simplex itself need not contain pv.)
166 // The return code is positive if we succeed (# of location steps at present)
167 // Return code of 0 means that we failed, perhaps because points with small weight have no Voronoi cells
168 // Return code of -1 means that we found the duplicate of a vertex in the mesh. IN THIS CASE, result
169 //  is the location of the corner where we found the duplicate!
170 int d3locSphere(const pd3stateType this, ppointType pv, int start,
171                 int *result) { // output
172   int j, guard; // loop variables
173   int c1, c2; // corners
174   psphereType s1, s2; // spheres
175   double I1, I2, d; // Insphere values
176
177   guard = 2*this->maxTetra+4; // prevent infinite loops
178   c1 = CORNER(start,0); // corner in start
179   s1 = this->sph+start; // sphere at start
180   I1 = InSpherev(s1, pv, this->s[c1+3].v); // Check if inside start sphere.
181   if ( (I1 < 0) || (I1 == 0 && s1->sq > 0) ){// found already
182     *result = start;
183     return 1; // success on first try
184   }
185
186   while (--guard) {
187     c2 = this->s[c1].opp; s2 = this->sph + TETRA(c2); // nhbr corner, sphere, value
188     I2 = InSpherev(s2, pv, this->s[LASTCORNER(c2)].v);
189     if ( (I2 < 0)  || (I2 == 0 && s2->sq > 0) ) {  // found one!
190       *result = s2 - this->sph;
191       return 2*this->maxTetra+5-guard; // number of steps
192     }
193     d = s2->sq * I1 - s1->sq * I2;   // Warning: if s1 & s2 are same sphere, this is zero
194     locateSideCnt++; // STATS
195     if (d==0) { // We are on a sphere---check for duplicate vertex
196       if (EQUALPV(pv,this->s[c2].v)) { *result = c2; return -1; }
197       j = INDEX(c2);
198       if (EQUALPV(pv,this->s[c2+offset[1][j]].v)) { *result = c2+offset[1][j]; return -1; }
199       if (EQUALPV(pv,this->s[c2+offset[2][j]].v)) { *result = c2+offset[2][j]; return -1; }
200       if (EQUALPV(pv,this->s[c2+offset[3][j]].v)) { *result = c2+offset[3][j]; return -1; }
201       // otherwise no duplicate; we probably have s1 == s2. (Rare in protein data.)
202       d = RANDBIT-0.5; // choose a random direction, as a hack. (Should do plane computation)
203     }
204     if (d < 0) // if on I1 side
205       c1 = INCREMENT(c1);
206     else {// on I2 side
207       c1 = INCREMENT(c2); s1 = s2; I1 = I2;
208     }
209   }
210   result = 0; // location failure
211   return 0;
212 }
213
214 /* d3compactCorners(const pd3stateType this, pcornerType *result, int *ncorners);
215  Takes corner table this->s in which some corners/tetrahedra are unused, and
216  returns a compactified corner table result, and its length ncorners.
217  DESTROYS the corner table s and active flags in the process.
218  returns false if it is unable to allocate memory.
219  */
220 int d3compactCorners(const pd3stateType this, // arrays this->s & this->active DESTROYED!
221                      pcornerType *result, int *ncorners) { // output
222   int c, nc, i, j;
223   pcornerType pc;
224
225   i = 0; // count actives
226   for (j = 0; j < this->maxTetra; j++)
227     this->active[j] = (DEAD(j))? -1 : i++; // make old->new pointer dictionary for active tetra
228
229   *ncorners = 4*i; // number of corners to return
230   *result = pc = (pcornerType) calloc(*ncorners, sizeof(cornerType)); // return corner table: v, opp
231   if (pc != NULL) {
232     for (j = 0; j < this->maxTetra; j++) // compact the corners
233       if (this->active[j] >= 0) {
234         c = CORNER(j,0);   // old corner c -->  new corner ptr pc = (*result)+0,1,2,...
235         pc->v = this->s[c].v; nc = this->s[c].opp;
236         pc->opp = CORNER(this->active[TETRA(nc)], INDEX(nc)); // assign new tetra # w/ old index
237         pc++; c++;
238         pc->v = this->s[c].v; nc = this->s[c].opp;
239         pc->opp = CORNER(this->active[TETRA(nc)], INDEX(nc));
240         pc++; c++;
241         pc->v = this->s[c].v; nc = this->s[c].opp;
242         pc->opp = CORNER(this->active[TETRA(nc)], INDEX(nc));
243         pc++; c++;
244         pc->v = this->s[c].v; nc = this->s[c].opp;
245         pc->opp = CORNER(this->active[TETRA(nc)], INDEX(nc));
246         pc++;
247       }
248   }
249   free(this->s);   // free old corner list
250   free(this->active);
251   return (pc != NULL);   // true if we were successful
252 }
253
254
255
```

Functions	
InSpherev	1:46
d3compactCorners	3:220
d3initialize	2:104
d3insert	4:262
d3locSphere	3:170
makeSphereV	1:52

```
256
257  /* d3insert(const pd3stateType this, int vi, int p)
258   inserts the point this->vert[vi] that is contained in sphere p
259   into the delaunay triangulation stored in this.
260   (d3locSphere may be used to obtain p.)
261   */
262  void d3insert(const pd3stateType this, int vi, int p) {
263    int i, j, off;
264    int b, c, newb; // corners
265    int nc, ni, dead, jdead; // indices
266    double d;
267    ppointType v0, v1, v2; // pointers to vertices
268    ppointType pv = this->vert+vi;
269
270      // Tetrahedra containing pv are "dead", and are pushed onto kill stack.
271      // We use DFS with stack pst to find them and kill them
272      // At live-dead boundary, we save dead tetras on stack nhbr,
273      //  then make new tetras and hook in to live by setting the last opp pointer.
274      //
275      // Invariants/operations: Tetrahedron p is marked alive or dead on first visit.
276      //    Corner c is pushed on stack when TETRA(this->s[c].opp) is marked dead.
277      //
278      // On termination, stack nhbr contains dead corners with live neighbors
279      //    that have new tetras (so this->s[nhbr].opp != this->s[this->s[nhbr].opp].opp temporarily.)
280      //    Stack kill contains old tetrahedra for final recycling.
281
282      stkINIT(this->dfs);  // DFS stack holds corners opposite dead tetras
283      stkINIT(this->idfs); // iDFS stack holds corners opposite infinite tetras with pv on bdry
284                           //    (these are a special case: dead, but don't propagate)
285      stkINIT(this->nhbr); // stack for dead corners with live nhbr tetras
286      stkINIT(this->kill); // stack of dead tetras to recycle
287      b = CORNER(p,0);
288      PUSH(p, this->kill); KILL(p); // kill tetra initial p,
289      PUSH(this->s[b++].opp, this->dfs); // stack neighbors
290      PUSH(this->s[b++].opp, this->dfs);
291      PUSH(this->s[b++].opp, this->dfs);
292      PUSH(this->s[b  ].opp, this->dfs);
293
294      while (!isEMPTY(this->dfs)) {
295        c = POP(this->dfs); p = TETRA(c);
296        /*     printf("::Popping %d with opp %d \n", c, this->s[c].opp);/**/
297        ASSERT(DEAD(TETRA(this->s[c].opp)), "dfs stack element with non-dead neighbor");
298        if (DEAD(p)) continue; // dead already
299        d = InSpherev(this->sph + p, pv, this->s[LASTCORNER(c)].v); // Is pv in, out, or on?
300        if (d < 0) { // kill and continue dfs if pv is strictly inside
301          KILL(p); PUSH(p, this->kill); // kill and stack tetra
302          j = INDEX(c);
303          PUSH(this->s[c+offset[1][j]].opp, this->dfs); // stack neighbors to check
304          PUSH(this->s[c+offset[2][j]].opp, this->dfs);
305          PUSH(this->s[c+offset[3][j]].opp, this->dfs);
306        }
307        else if (d > 0 || this->sph[p].sq > 0) { // pv is outside (or on with sp finite), so
308                                                 // c is live neighbor of dead opp tetra this->s[c].opp
309          PUSH(this->s[c].opp, this->nhbr); // remember old corner, so we can hook tetra into mesh later
310          STARTTETRA(this); // make new tetrahedron liveTetra
311          newb = CORNER(this->liveTetra,3); // last corner of new tetra
312          setCornerVCN(newb, pv, c); // last corner is pv; also set opposite corner c. Do rest later.
313        }
314          else { // d==0 && sph[p] is infinite: handle two special cases
315            if (this->sph[TETRA(this->s[c].opp)].sq == 0) { // if dead sphere is infinite, too
316              PUSH(c, this->idfs); // then if c stays alive, we make tetra to it (flat, but infinite).
317            } else { // dead sphere is finite; kill c and make tetras to neighbors, if they stay alive.
318              KILL(p); PUSH(p, this->kill); // kill and stack tetra
319              j = INDEX(c);
320              PUSH(this->s[c+offset[1][j]].opp, this->idfs); // stack neighbors to check
321              PUSH(this->s[c+offset[2][j]].opp, this->idfs);
322              PUSH(this->s[c+offset[3][j]].opp, this->idfs);
323            }
324          }
325      }
326
327      while (!isEMPTY(this->idfs)) { // check the neighbors of infinite tetrahedra
328        c = POP(this->idfs); p = TETRA(c);
329        /*     printf("::Popping %d with opp %d \n", c, this->s[c].opp);/**/
330        ASSERT(DEAD(TETRA(this->s[c].opp)), "dfs stack element with non-dead neighbor");
331        if (DEAD(p)) continue; // dead already
332        ASSERT(DEAD(TETRA(this->s[c].opp)), "Live corner c should have dead neighbor");
333        PUSH(this->s[c].opp, this->nhbr); // remember old corner, so we can hook tetra into mesh later
334        STARTTETRA(this); // make new tetrahedron liveTetra
335        newb = CORNER(this->liveTetra,3); // last corner of new tetra
336        setCornerVCN(newb, pv, c); // last corner is pv; also set opposite corner c. Do rest later.
337      }
338
339      // Now, we have stack of dead neighbors of live tetras, and we've hooked new tetras to them.
340      while (!isEMPTY(this->nhbr)) {
341        dead = POP(this->nhbr); jdead = INDEX(dead); //  dead tetra and index of dropped corner.
342        /*     printf("--Popped %d(%d)\n", dead, jdead); /**/
343        ASSERT(DEAD(TETRA(dead)), "corner on nhbr stack is not dead!?");
344        newb = this->s[this->s[dead].opp].opp-3; // base of new tetra.
345
346        dead -= jdead; // just use base of dead one.
347        // new tetra has 0,1,2,3=pv;
348        // corresponding old indices before jdead is dropped:
349        //  drop[j][0],..,drop[j][3], (no corresp to pv)
350        j = jdead;
```

Functions	
InSpherev	1:46
d3compactCorners	3:220
d3initialize	2:104
d3insert	4:262
d3locSphere	3:170
makeSphereV	1:52

```
351         i = drop[jdead][0]; // old index of new corner 0;
352         c = dead+i; // note i = INDEX(C);
353         v0 = this->s[c].v; // copy vertex v0
354         nc = this->s[c].opp; // go to neighbor
355         // In tetra opp c, find new location of j.  That is new c. New j = INDEX(c.opp).
356         // To avoid index calculations, maintain i = INDEX(c), nc = this->s[c].opp, ni = INDEX(nc).
357         while (DEAD(TETRA(nc))) {
358           ni = INDEX(nc); off = indoff[i][ni][j]; // where j goes relative to i is our new i.
359           j = ni; i = ni + off; c = nc + off; nc = this->s[c].opp; // fix new j, i, c, and try neighbor
360         }
361         nc = this->s[nc].opp; // go to new tetra
362         ASSERT(this->s[nc].v == pv, "Expected to find new tetra using pv after walking dead tetras. ");
363
364         setCornerVC(newb, v0, nc-3+invdrop[i][j]); newb++;
365
366         j = jdead;
367         i = drop[jdead][1]; // old index of new corner 1;
368         c = dead+i; // note i = INDEX(C);
369         v1 = this->s[c].v; // copy vertex v1
370         nc = this->s[c].opp; // go to neighbor
371         // In tetra opp c, find new location of j.  That is new c. New j = INDEX(c.opp).
372         // To avoid index calculations, maintain i = INDEX(c), nc = this->s[c].opp, ni = INDEX(nc).
373         while (DEAD(TETRA(nc))) {
374           ni = INDEX(nc); off = indoff[i][ni][j]; // where j goes relative to i is our new i.
375           j = ni; i = ni + off; c = nc + off; nc = this->s[c].opp; // fix new j, i, c, and try neighbor
376         }
377         nc = this->s[nc].opp; // go to new tetra
378         ASSERT(this->s[nc].v == pv, "Expected to find new tetra using pv after walking dead tetras. ");
379
380         setCornerVC(newb, v1, nc-3+invdrop[i][j]); newb++;
381
382         j = jdead;
383         i = drop[jdead][2]; // old index of new corner 2;
384         c = dead+i; // note i = INDEX(C);
385         v2 = this->s[c].v; // copy vertex v2
386         nc = this->s[c].opp; // go to neighbor
387         // In tetra opp c, find new location of j.  That is new c. New j = INDEX(c.opp).
388         // To avoid index calculations, maintain i = INDEX(c), nc = this->s[c].opp, ni = INDEX(nc).
389         while (DEAD(TETRA(nc))) {
390           ni = INDEX(nc); off = indoff[i][ni][j]; // where j goes relative to i is our new i.
391           j = ni; i = ni + off; c = nc + off; nc = this->s[c].opp; // fix new j, i, c, and try neighbor
392         }
393         nc = this->s[nc].opp; // go to new tetra
394         ASSERT(this->s[nc].v == pv, "Expected to find new tetra using pv after walking dead tetras. ");
395
396         setCornerVC(newb, v2, nc-3+invdrop[i][j]); newb++;
397
398         c = this->s[this->s[dead+jdead].opp].opp;
399         ASSERT(v0==this->s[CORNERINOPP(0,c)].v, "v0 does not line up");
400         ASSERT(v1==this->s[CORNERINOPP(1,c)].v, "v1 does not line up");
401         ASSERT(v2==this->s[CORNERINOPP(2,c)].v, "v2 does not line up");
402
403         makeSphereV(this->sph+TETRA(newb), v0, v1, v2, pv, this->vert); // use either this or makeSphereP abov
    e
404       }
405     }
406
407
408
```

Functions	
InSpherev	1:46
d3batch	6:416
d3compactCorners	3:220
d3initialize	2:104
d3insert	4:262
d3locSphere	3:170
makeSphereV	1:52

```
409
410 /* d3.c Delaunay/Power diagram function
411  d3batch takes lifted input vertices, and returns a (compact) corner table for Delaunay.
412  REQUIRES that the first point is at infinity,
413  and that the first 5 are in general position. (I should verify or relax this.)
414  Since all points have radii assigned already, it can compute power diagrams.
415  */
416 void d3batch(ppointType vertArray, int nvert, // input vertices (pointType[]) & number of vertices
417         pcornerType *result, int *ncorners)   // output corner table
418 {
419   int k;
420   int p; // tetra
421   int vi; // vertex index in outermost loop
422   ppointType pv; // vertex pointer in outermost loop
423   pcornerType pc; // corner for copying to result
424   d3stateType d3; // state data structure
425   const pd3stateType this = &d3;
426
427   d3initialize(this, vertArray, nvert);
428 #ifdef STATS
429   this->killmax = this->dfsmax = this->idfsmax = this->nhbrmax = -1; // init STATS
430 #endif
431   maxLocate = locateSideCnt = sphereCnt = startTetraCnt = freeTetraCnt= inSphereCnt=0;
432
433   for (vi = 5; vi < nvert; vi++) { // incrementally insert vert[vi]
434     //LOCATE: find some tetrahedron with sphere strictly containing vert[vi]
435     if ((k = d3locSphere(this, this->vert+vi, this->liveTetra, &p)) < 1)
436       continue; // if we fail to locate (duplication or other reason) just skip vert[vi]
437     if (k > maxLocate) maxLocate = k; // STATS
438 #ifndef NOASSERT
439     if (vi%100 == 0) { // print a few search results
440       printf("%%Found %3d in %4d(%3d) after %d steps %f\n",
441              vi, CORNER(p,0), p, k, InSpherev(this->sph+p, this->vert+vi, this->s[CORNER(p,3)].v));
442       auditCorners(this, 1); // audit, and check spheres, too
443       (void)fflush(stdout);
444     } /**/
445 #endif
446     d3insert(this, vi, p); // insert vertex vi, which is in sphere of tetra p.
447     while (!isEMPTY(this->kill)) { // recycle memory of dead tetrahedra
448       p = POP(this->kill);
449       FREETETRA(this, p);
450     }
451   }
452 #ifdef STATS
453   printf(" %10d\t Points\n %10d\t Max Tetra\n", vi, this->maxTetra);
454       printf("We performed:\n %10d\tinSphere tests\n %10d\tplane tests\n %10d\ttetrahedra created-\n %10d\tf
    reed =\n %10d\ttetrahedra\n %10d\tsphere equations computed.\n",
455              inSphereCnt, locateSideCnt, startTetraCnt, freeTetraCnt, startTetraCnt-freeTetraCnt, sphereCnt)
    ;
456       printf("Max locate steps=%d, max killed=%d, max inf=%d, max created=%d, maxdfs=%d\n", this->maxTetra,
457              maxLocate, this->killmax, this->idfsmax, this->nhbrmax, this->dfsmax);
458       fflush(stdout);
459       auditCornersAux(this,1); /*TESTING*/
460 #endif
461
462   free(this->sph);   // done with spheres; free them
463
464   if (!d3compactCorners(this, result, ncorners)) { // disposes s and active!!
465     printf("ERROR: d3.c could not allocate corner table to return\n");
466     exit(EXIT_FAILURE);
467   }
468 }
469
```

Functions	
main	1:9

```
1   /* test.c              Jack Snoeyink Oct 2003
2    test program for d3.c, d3permute.c
3    */
4
5   #include <time.h>
6   #include "delaunay3.h"
7   #include "pdbreader.h"
8
9   int main(int argc, char** argv) {
10    int i, j, ii, jj;
11    int hashtb[HIGHCOORD];
12    int nexttb[NVERT];
13    ppointType vert,v;
14    int nvert, nv;
15    pcornerType result;
16    int nresult;
17
18    clock_t tic, toc;
19
20    FILE *fid;
21
22    /*  fid = rndopener("none", argc, argv);
23     d3permute(fid, rndreader, &vert, &nvert);
24     fclose(fid);/**/
25
26    /* fid = txtopener("hilb6.txt", argc, argv);
27     d3permute(fid, txt3reader, &vert, &nvert);
28     fclose(fid); /**/
29
30    ii = 1;
31    for (ii = 1; ii < argc; ii++)
32      if (argv[ii][0] != '-') {
33        fid = pdbopener(argv[ii], argc<4 ? argc : 4, argv);
34        d3permute(fid, pdbreader, &v, &nv);
35        fclose(fid);
36        tic = clock(); // start timer
37        jj = 1;
38  #ifdef NOASSERT
39        for (jj = 0; jj < 10; jj++)
40  #endif
41          {
42            d3batch(v, nv, &result, &nresult);
43            free(result);
44          }
45        toc = clock();
46        printf("%s %d %d Time(secs) %f\n\n", argv[ii], nv, nresult,
47               ((double) (toc - tic)) / CLOCKS_PER_SEC / jj); (void)fflush(stdout); /**/
48      }
49    return EXIT_SUCCESS;
50  }
51
```