```
#define qh_QHimport
#include "qhull_ra.h"
#include "result.h"

double* inverse3X3(double mat){
double det = det3_(mat[0][0], mat[0][1], mat[0][2],
mat[1][0], mat[1][1], mat[1][2],
mat[2][0], mat[2][1], mat[2][2]);
double d00 = det2_(mat[1][1], mat[1][2], mat[2][1], mat[2][2]);
double d01 = det2_(mat[1][0], mat[1][2], mat[2][0], mat[2][2]);
double d02 = det2_(mat[1][0], mat[1][1], mat[2][0], mat[2][1]);
double d10 = det2_(mat[0][1], mat[0][2], mat[2][1], mat[2][2]);
double d11 = det2_(mat[0][0], mat[0][2], mat[2][0], mat[2][2]);
double d12 = det2_(mat[0][0], mat[0][1], mat[2][0], mat[2][1]);
double d20 = det2_(mat[0][1], mat[0][2], mat[1][1], mat[1][2]);
double d21 = det2_(mat[0][0], mat[0][2], mat[1][0], mat[1][2]);
double d22 = det2_(mat[0][0], mat[0][1], mat[1][0], mat[1][1]);
static double out[3][3];
out[0][0] = d00/det;
out[0][1] = -d10/det;
// out = { { d00/det, -d10/det, d20/det}
// , {-d01/det, d11/det, -d21/det}
// , { d02/det, -d12/det, d22/det} };
return out;
}

double** solve3X3(double** mat, double* vec){
double*** imat = inverse3X3(mat);
static double out[3];
out = { imat[0][0]vec[0]+imat[0][1]vec[1]+imat[0][2]vec[2] , imat[1][0]vec[0]+imat[1][1]vec[1]+imat[1][2]vec[2]
, imat[2][0]vec[0]+imat[2][1]vec[1]+imat[2][2]*vec[2] };
return out;
}

struct Result* delaunay(
double* vertices,
unsigned dim,
unsigned n,
unsigned* nf,
unsigned* exitcode,
char* tmpFile
)
{
char flags[250]; /* option flags for qhull, see qh_opt.htm/ sprintf(flags, "qhull d Qt Fn Qbb", ""); qhT qh_qh; / Qhull's data
structure. First argument of most calls */
qhT *qh= &qh_qh;
QHULL_LIB_CHECK
boolT ismalloc = False; /* True if qhull should free points in qh_freeqhull() or reallocation */
FILE *errfile = NULL;
int curlong, totlong; /* to free the memory later / unsigned indices;
double* areas;
// int* sizneighbors;
unsigned* neighbors;
double* centers;
unsigned* toporient;
unsigned** ridges; // size n_neighbors X (1+dim) ; first column for id
```

```
unsigned n_neighbors;

// FILE* tmpstdout = fopen(tmpFile, "w");
FILE* tmpstdout = tmpfile();
exitcode[0] = qh_new_qhull(qh, dim, n, vertices, ismalloc, flags, tmpstdout,
errfile);
fclose(tmpstdout);
FILE* summaryFile = fopen(tmpFile, "w");
qh_printsummary(qh, summaryFile);
fclose(summaryFile);
qh_getarea(qh, qh->facet_list);
if (!exitcode[0]) { /* 0 if no error from qhull */
facetT *facet; /* set by FORALLfacets */
vertexT *vertex, **vertexp;
facetT *neighbor, **neighborp;
// ridgeT *ridge, **ridgep;
int numfacets = qh->num_facets;
//coordT *center, *centerp; / Count the number of facets so we know how much space to allocate / nf[0]=0; / Number of facets / //
int k = malloc(1+sizeof(int)qh->num_facets); // k[0] = 0; // int l =0; //int m =0; // int facetsok = malloc(qh->num_facets * sizeof(int));
// int* facetsid = malloc(qh->num_facets * sizeof(int));

// FORALLfacets {
// // if (!facet->upperdelaunay && facet->simplicial && !facet->degenerate) {
// unsigned delete = 0;
// FOREACHvertex_(facet->vertices) {
// FOREACHneighbor_(vertex) {
// if (!qh_setin(facet->neighbors, neighbor)){ //!qh_setin(vertex->neighbors, neighbor)) {
// delete++;
// printf("delete vertex %d in facet %d\n", vertex->id, facet->id);
// qh_removevertex(qh, vertex);
// }
// }
// // FOREACHneighbor_(vertex) {
// // if (!qh_setin(facet->neighbors, neighbor)) {
// // delete++;
// // printf("delete ---neighbor---- facet %d\n", facet->id);
// // qh_removefacet(qh,facet);
// // }
// // }
// // FOREACHneighbor_(vertex) {
// // FOREACHridge_(neighbor->ridges) {
// // if (!qh_setin(facet->ridges, ridge)) {
// // delete++;
// // printf("delete vertex BECAUSE PB RIDGE %d\n", vertex->id);
// // qh_removevertex(qh, vertex);
// // }
// // }
// // }
// }
// if(delete){
// printf("DON'T DELETE FACET %d", facet->id);
// //qh_removefacet(qh,facet);
// }
// //}
// }
```

```
    int* facetsvisitid = malloc(numfacets * sizeof(int));
    FORALLfacets {
```

```
// qh_makeridges(qh, facet);
// facetsok[facet->id] = 1;
//printf("visitid: %d - id: %d\n", facet->visitid, facet->id);
if (!facet->upperdelaunay && facet->simplicial && !facet->degenerate) {
nf[0]++;
facet->id = nf[0];
facetsvisitid[nf[0]] = facet->visitid;
// FOREACHridge_(facet->ridges) {
// printf("facet %d top ridge: %d\n", facet->id, ridge->top->id);
// printf("facet %d bottom ridge: %d\n", facet->id, ridge->bottom->id);
// }
// facetsid[facet->id] = (int)(nf[0]);
//printf("orientation: %d ", facet->toporient);
// facet->id = l>0 && facet->id > 0 && facet->id <=m &&="" k[facet-="">id-1]>0 ? facet->id - k[facet->id-1] : facet->id;
// for(int k=0; knormal[k]);
// printf("\n");
// }
}else{
// facetsok[facet->id] = 0;
// facetsid[facet->id] = 0;
qh_removefacet(qh, facet);
// l++;
// il faut réindexer les neighbours !
}
//m++;
// k[m] = l;
// printf("%d", l);
}
// vertexT* vertexx = qh->vertex_list;
// w hile(vertexx && vertexx->next){
// //if(qh_setsize(qh, vertexx->neighbors)){
// qh_order_vertexneighbors(qh, vertexx);
// vertexx = vertexx->next;
// }

// qh->facet_list = qh->new facet_list;
// faire une nouvelle liste de facettes pour que les neighbours soient ok
/* Alocate the space / indices = (unsigned) malloc(nf[0] * (dim+1) * sizeof(unsigned));
areas = (double*) malloc(nf[0] * sizeof(double));
//sizneighbors = (int*) malloc(nf[0] * sizeof(int));
neighbors = (unsigned*) malloc(nf[0] * (dim+1) * sizeof(unsigned));
centers = (double*) malloc(nf[0] * dim * sizeof(double));
toporient = (unsigned*) malloc(nf[0] * sizeof(unsigned));
unsigned* neighborok = malloc(nf[0] * sizeof(unsigned));
n_neighbors = 0;
```

```
/* Iterate through facets to extract information - first pass */
    unsigned i = 0; // facet counter
FORALLfacets {
        unsigned j;

        areas[i] = facet->f.area;
        j = 0;
  FOREACHvertex_(facet->vertices) {
    indices[i*(dim+1)+j] = qh_pointid(qh, vertex->point);
    j++;
        }

        j = 0;
        FOREACHneighbor_(facet) {
            //sizneighbors[i] = qh_setsize(qh, facet->neighbors);
            //printf("%d\n", sizneighbors[i]); // toujours dim+1
            //printf("visitid: %d - id: %d\n", neighbor->visitid, neighbor->id); // ? neighbor->visitid: 0 - neighbor->id)
            //neighbors[i*(dim+1)+j] = neighbor->visitid ? (unsigned)neighbor->visitid: (unsigned)0;
            // neighbors[i*(dim+1)+j] = neighbor->visitid >= numfacets || neighbor->id > nf[0] ? // marche pas pour test4
            // (unsigned)(0) : (unsigned)(neighbor->id);
            // OK: neighbors[i*(dim+1)+j] = neighbor->id > nf[0] ? // marche pas pour test4 !
            //                                              (unsigned)(0) :
            //                                              (neighbor->visitid == facetsvisitid[neighbor->id] ?
            //                              (unsigned)(neighbor->id) : 0);
            if(neighbor->id > nf[0] || neighbor->visitid != facetsvisitid[neighbor->id]){
                neighbors[i*(dim+1)+j] = (unsigned)0;
                neighborok[neighbor->id] = 0;
            }else{
                neighbors[i*(dim+1)+j] = (unsigned)(neighbor->id);
                neighborok[neighbor->id] = 1;
                n_neighbors++;
            }
            //neighbors[i*(dim+1)+j] = facetsok[neighbor->id] ? (unsigned)(facetsid[i]) : (unsigned)(0);
            j++;
        }

        i++;
    }

    double* distances = malloc(n_neighbors * sizeof(double));
    ridges = malloc(n_neighbors * sizeof(unsigned*)); // (intersections, adjacencies, ridges)
    //unsigned* neighborsIndices = malloc(n_neighbors * sizeof(unsigned));
    // unsigned combinations[4][3] = { {0, 1, 2}
  //                               , {0, 1, 3}
    //                                                        , {0, 2, 3}
    //                                                        , {1, 2, 3} };

    unsigned i_neighbor = 0;
    i = 0; // facet counter
FORALLfacets {

        toporient[i] = facet->toporient;

        coordT* center = qh_facetcenter(qh, facet->vertices);
        for(unsigned j=0; j<dim; j++){
            centers[i*dim+j] = center[j];
        }

        FOREACHneighbor_(facet){
            if(neighborok[neighbor->id]){

                double dist;
                qh_distplane(qh, center, neighbor, &dist);
                printf("dist to center %d (=%d) to neighbor %d: %f (negative ? %d ; zero ? %d)\n",
                       facet->id, i, neighbor->id, dist, dist<0, dist==0);

                // vaudrait mieux un tableau 3d - non va y avoir des vides
                //neighborsIndices[i_neighbor] = neighbor->id;
                // on pourrait mettre le id dans la 1ère colonne
                ridges[i_neighbor] = (unsigned*) malloc((1+3)*sizeof(unsigned));
                ridges[i_neighbor][0] = neighbor->id; // plutôt facet id
                unsigned* combination = malloc(3*sizeof(unsigned));
                unsigned k = 0;
                FOREACHvertex_(neighbor->vertices) {
                    unsigned vertexid = qh_pointid(qh, vertex->point);
                    if (vertexid == indices[i*(dim+1)+0] ||
                          vertexid == indices[i*(dim+1)+1] ||
                            vertexid == indices[i*(dim+1)+2] ||
                            vertexid == indices[i*(dim+1)+3])
                    { // je crois que ce test peut planter : s'il n'y a pas dim adjacences... non dans ce cas c'est pas un
```

```
                        unsigned l = 0;
                        while(1){
                            if(vertexid == indices[i*(dim+1)+l]){
                                break;
                            }
                            l++;
                        }
                        combination[k] = l;
                        ridges[i_neighbor][k+1] = vertexid;
                        k++;
                    }
                }
                printf("facet id: %d\n", facet->id);
                vertexT* vertex1 = (vertexT*)facet->vertices->e[combination[0]].p;
                vertexT* vertex2 = (vertexT*)facet->vertices->e[combination[1]].p;
                vertexT* vertex3 = (vertexT*)facet->vertices->e[combination[2]].p;
                printf("vertex1: %d - ", qh_pointid(qh, vertex1->point));
                printf("vertex2: %d - ", qh_pointid(qh, vertex2->point));
                printf("vertex3: %d \n ", qh_pointid(qh, vertex3->point));
                pointT* point1 = vertex1->point;
                pointT* point2 = vertex2->point;
                pointT* point3 = vertex3->point;
                double u1 = point2[0]-point1[0];
                double v1 = point2[1]-point1[1];
                double w1 = point2[2]-point1[2];
                double u2 = point3[0]-point1[0];
                double v2 = point3[1]-point1[1];
                double w2 = point3[2]-point1[2];
                // faudrait aussi retourner la normale ou alors résoudre le système ici et retourner le vecteur
                double* normal = malloc(3*sizeof(double));
                normal[0] = det2_(v1, v2, w1, w2);
                normal[1] = det2_(u2, u1, w2, w1);
                normal[2] = det2_(u1, u2, v1, v2);
                qh_normalize2(qh, normal, 3, 1, NULL, NULL); // 3:dim 1:toporient
                double offset = -(point1[0]*normal[0]+point1[1]*normal[1]+point1[2]*normal[2]);
                distances[i_neighbor] = qh_distnorm(3, center, normal, &offset);
                double mat[3][3] = { {u1, v1, w1}
                                                     , {u2, v2, w2}
                                                   , {normal[0], normal[1], normal[2]} };
                double rhs[3] = { center[0]*u1 + center[1]*v1 +center[2]*w1
                                                , center[0]*u2 + center[1]*v2 +center[2]*w2
                                              , -offset };
                double* vector = solve3X3(mat, rhs);
                printf("Vertex1: %f %f %f\n", point1[0], point1[1], point1[2]);
                printf("Vertex2: %f %f %f\n", point2[0], point2[1], point2[2]);
                printf("Vertex3: %f %f %f\n", point3[0], point3[1], point3[2]);
                printf("NORMAL: %f %f %f\n", normal[0], normal[1], normal[2]);
                printf("MYDISTANCE: %f\n", mydistance);
                printf("VECTOR: %f %f %f\n", vector[0], vector[1], vector[2]);
                free(vector);
                i_neighbor++;
                // ? free(combination);
            }
        }
        i++;
    }

    free(facetsvisitid);
    free(neighborok);
    //free(facetsok);
    //free(facetsid);
    // qhT myqh_qh;
    // qhT *myqh= &myqh_qh;
    // myqh->hull_dim = 3;
    // myqh->RANDOMdist=0;
    // myqh->IStracing=0;
    // myqh->MINdenom = 1e-16;
    // myqh->MINdenom_1 = 1e-16;
    // facetT* thefacet = qh->facet_list;
    // coordT* thecenter = qh_facetcenter(qh, thefacet->vertices);
  // facetT* myfacet = malloc(sizeof(facetT));
    // setT* myvertices = qh_setnew(myqh, 3);
    // vertexT* vertex1 = (vertexT*)thefacet->vertices->e[0].p; // indices[0*(dim+1)+0]
    // vertexT* vertex2 = (vertexT*)thefacet->vertices->e[1].p;
```

```
        // vertexT* vertex3 = (vertexT*)thefacet->vertices->e[2].p;
        // qh_setappend(myqh, &myvertices, (void*)vertex1);
        // qh_setappend(myqh, &myvertices, (void*)vertex2);
        // qh_setappend(myqh, &myvertices, (void*)vertex3);
        // myfacet->vertices = myvertices;
        // pointT* point1 = vertex1->point;
        // pointT* point2 = vertex2->point;
        // pointT* point3 = vertex3->point;
        // double u1 = point1[1]-point1[0];
        // double u2 = point2[1]-point2[0];
        // double u3 = point3[1]-point3[0];
        // double v1 = point1[2]-point1[0];
        // double v2 = point2[2]-point2[0];
        // double v3 = point3[2]-point3[0];
        // double* normal = malloc(3*sizeof(double));
        // normal[0] = det2_(u2, v3, u3, v2);
        // normal[1] = det2_(u3, v1, u1, v3);
        // normal[2] = det2_(u1, v2, u2, v1);
        // //qh_normalize2(myqh, normal, 3, 1, NULL, NULL); // 3:dim 1:toporient
        // double offset = -(point1[0]*normal[0]+point1[1]*normal[1]+point1[2]*normal[2]);
        // myfacet->normal = normal;
        // myfacet->offset = offset;
        // // double mydistance;
        // // qh_distplane(myqh, thecenter, myfacet, &mydistance);
        // // printf("MYDISTANCE: %f", mydistance);
        // free(myfacet);
        // int mycurlong, mytotlong;
        // qh_freeqhull(myqh, !qh_ALL);                  /* free long memory */
        // qh_memfreeshort (myqh, &mycurlong, &mytotlong);   /* free short memory and memory allocator */
    }



    struct Result* out = malloc(sizeof(ResultT));
    if(!exitcode[0]){
        unsigned* ridges_ = malloc(n_neighbors*(1+3)*sizeof(unsigned));
        for(unsigned l=0; l<n_neighbors; l++){
            printf("l=%d:\n", l);
            for(unsigned ll=0; ll<4; ll++){
                ridges_[l*(1+dim)+ll] = ridges[l][ll];
                printf("ll=%d - id=%d ", ll, ridges_[l*dim+ll]);
            }
        }
      out->dim       = dim;
      out->length    = nf[0];
      out->indices   = indices;
        out->areas     = areas;
        out->neighbors = neighbors;
        out->centers   = centers;
        out->toporient = toporient;
        out->ridges    = ridges_;
    }

    free(ridges); // remettre ça avant
    /* Do cleanup regardless of whether there is an error */
    qh_freeqhull(qh, !qh_ALL);                  /* free long memory */
    qh_memfreeshort (qh, &curlong, &totlong);   /* free short memory and memory allocator */
    // if (exitcode) {
    //  error("Received error code %d from qhull.", exitcode);
    // }
```

return out;

}