TP1

```
void projection(Vec3 input , Vec3 & output ,Vec3 const & position , Vec3 const & normal){
    float X  = Vec3::dot(  ( input - position ) , normal) / normal.length();
    output = input -  X* normal;
}

void HPSS( Vec3 inputPoint , Vec3 & outputPoint , Vec3 & outputNormal ,
 std::vector<Vec3>const & positions , std::vector<Vec3>const & normals , BasicANNkdTree const & kdtree ,
 int kernel_type, unsigned int nbIterations = 1 , unsigned int knn = 10 ) {
 int k=0;
 while(k<nbIterations){
   ANNidxArray id_nearest_neighbors =new ANNidx[ knn ];
   ANNdistArray square_distances_to_neighbors = new ANNdist[ knn ];

   kdtree.knearest( inputPoint , knn , id_nearest_neighbors, square_distances_to_neighbors );

   Vec3 n_nomi = Vec3(0,0,0);
   Vec3 c_nomi=  Vec3(0,0,0);
   float sumW=0;
   Vec3 output[knn];

   for( int i=0; i<knn; i++){
    projection(inputPoint, output[i], positions[id_nearest_neighbors[i]], normals[id_nearest_neighbors[i]]);
    float h =  sqrt(square_distances_to_neighbors[knn-1]);
    double w=0;
    double r = (inputPoint - positions[id_nearest_neighbors[i]]).length();
    if (kernel_type==0){
        w=exp(-pow(r,2)/pow(h,2));
    }
        if (kernel_type==1){
        w=pow(1- ( r / h ) ,4) * ( 1 + 4 * ( r / h ) );
    }
    if (kernel_type==2){
        w=pow(h/r,2);
    }
    c_nomi += w*output[i];
    n_nomi += (w*normals[id_nearest_neighbors[i]]);
    sumW += w;
  }
  outputPoint =  c_nomi / sumW;
  outputNormal = n_nomi / sumW;
  delete [] id_nearest_neighbors;
  delete [] square_distances_to_neighbors;
  k++;
  inputPoint = outputPoint;
}
}
```

TP3

```
void centroide(std::vector<Vec3>const & position,  Vec3 & centroide)
{
    for(int i =0; i< position.size();i++){
        centroide+=position[i];}
    centroide= (1.0/position.size())*centroide;
    printf(" ll %f, %f, %f \n",centroide[0],centroide[1],centroide[2] );
}
```

```
void translation(std::vector<Vec3>& position,  Vec3 & source, Vec3 & target)
{
   Vec3 translation = target - source;
   for(int i =0; i< position.size();i++){
      position[i]+= translation;
   }
}

void PositionToCentroide(std::vector<Vec3> & position, std::vector<Vec3> & out,Vec3 & target) {
    out.resize( position.size() );
   for(int i =0; i< position.size();i++){
      Vec3 translation = target-position[i];
      out[i]= translation;
     // printf("lala %f  \n",position[i][0]);
   }
}


//1_ calculs du centroide
//2_ translate source sur target
//3_ rotation aleatoire (ou l'Identité)
//2_ et 3_ -> calculs de l'initialisation = ACP -> recallé les axes (ou pas) # init
//ICP
void ICP( std::vector<Vec3>  &ps , std::vector<Vec3> const &nps ,
         std::vector<Vec3>  &qs , std::vector<Vec3> const & nqs ,
         BasicANNkdTree const & qsKdTree , Mat3 & rotation , Vec3 & translation ,
         unsigned int nIterations ) {
   int ite=0;
while(ite++<nIterations){
      centroide(ps, Centroide1);
      centroide(qs, Centroide2);
      std::vector<Vec3> pslocal;
      std::vector<Vec3> qslocal;
      PositionToCentroide( ps, pslocal , Centroide2);
      std::vector<Vec3> psNearest;
      psNearest.resize( ps.size() );
      for(int i =0; i< ps.size(); i++){
         psNearest[i]=ps[qsKdTree.nearest(ps[i])];
      }
      PositionToCentroide( psNearest,  qslocal,Centroide2);
      Mat3 S = Mat3();
      for(int i =0; i<3; i++)
         for( int j=0; j<3; j++){
            float val=0;
            for(int k = 0; k< pslocal.size(); k++){
               val += pslocal[k][i]*qslocal[k][j];
               //  printf("lala %f  %f\n",val ,pslocal[k][i],qslocal[k][j]);
            }
         S(i,j)= val;
         }
      S.setRotation();
      for(int i =0; i< ps.size(); i++){
         ps[i] = Centroide2 + S * (ps[i] - Centroide1);
      }
      }
}
```