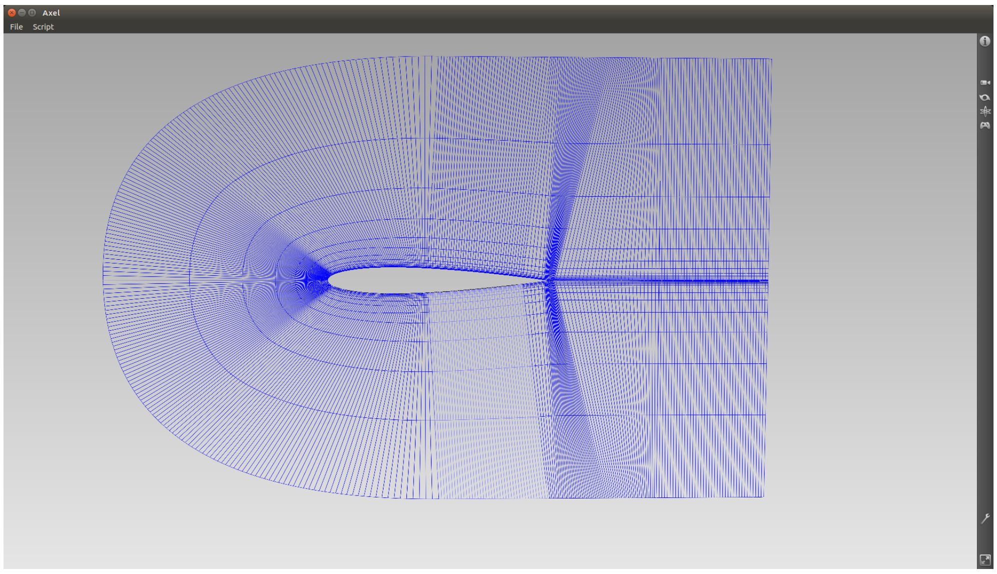
Construction of analysis suitable mesh for the treatment of potential flow framework based on Spline surfaces

Techical report

Contents

1 Domain overview



The goal of this work is to extract analysis suitable via parametrizing B-spline surfaces for the computation of potential flow problems around air foil.

Thus, for computational reasons we are interested in constructing a denser mesh (surfaces with denser knot line distribution by consequence) particularly int the following regions of the air foil:

- the trailing edge
- the leading edge
- Boundary around the air foil

In what follows we give a description in the algorithms invoked for the completion of this goal.

2 subdivision algorithm

Let $s(t) \equiv s(u(t), v(t))$ and $k(\tau) \equiv k(u(\tau), v(\tau))$ denote the curve representing the foil and contour curve segment respectively.A subdivision algorithm(***Listing 1***) located in **get_meshes.hpp** was put into place in order to obtain a manipulative distribution of points $P_i(x(t_i), y(t_i)) \in s(t)$,taking into consideration the curvature of the air foil and the euclidean distance of each point from the trailing edge .During the subdivision algorithm new point are selected applying the formula(***Listing 1,Line 28***):

$$Q_i^n = \frac{P_{i-1}^{n-1} + P_i^{n-1}}{2}$$

where $P_{i-1}^{n-1}, P_i^{n-1} \in s(t)$ and Q_i^n is the mid point of the line segment formed by those two points .By projecting Q_i^n to $s(t)$ we acquire the next point P_i^n(***Listing 1,Line 30***) .

Listing 1: subdivision algorithm

```
1  void get_meshes_contour( axlAbstractCurveBSpline* axlcurve, axlAbstractCurveBSpline*
       contour_curve,axlMesh* physical_mesh,axlMesh* parameter_mesh,const double
       points_on_line, double points_on_curve)
2  ...
3  //subdivision algorithm for distribution of point along the
4    airfoil
5      std::vector<axlPoint> P;
6      P.push_back(P0);
7      P.push_back(P1);
8      std::map<int,int> edge;
9      edge[0]=1;
10     int b,e;
11     std::list<int> stack;
12     stack.push_front(0);
13     while(!stack.empty())
14     {
15
16         std::list<int>::iterator it=stack.begin();
17         b=*it;
18         stack.pop_front();
19         e=edge[b];
20         axlPoint P_b=P.at(b);
21         axlPoint P_e=P.at(e);
22
23         double u_b=dst.nearestCurveParameter(P_b.x(),P_b.y(),P_b.z(),i);
24         double u_e=dst.nearestCurveParameter(P_e.x(),P_e.y(),P_e.z(),i);
25
26         if (axlPoint::distance(P_b,P_e)*alt_distance_factor(P_b,P_e)*curvature_factor(
               axlcurve,u_b,u_e)>(1/points_on_curve))
27         {
28             axlPoint P_middle_on_line=(P.at(b)+P.at(e))/2;
29             double u_projected=dst.nearestCurveParameter(P_middle_on_line.x(),
                   P_middle_on_line.y(),P_middle_on_line.z(),i);
30             axlPoint P_middle_projected=axlcurve->eval(u_projected);
31             P.push_back(P_middle_projected);
32             int n=P.size()-1;
33             edge[b]=n;
```

```
                edge[n]=e;
                stack.push_back(b);
                stack.push_back(n);
            }

    .... }
```

The subdivision algorithm takes into consideration the following criteria(***Listing 1,Line 26***):

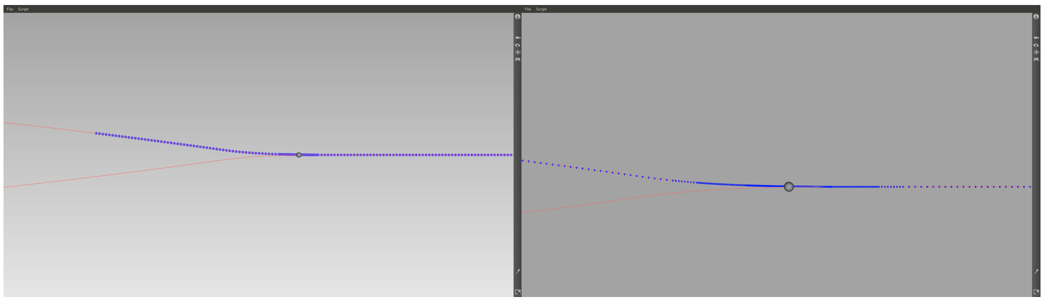
```
if $(d(P_{i-1}^{n-1}, P_i^{n-1}) * f_{dist}(d(P_{i-1}^{n-1}, A), d(P_i^{n-1}, A)) * g_{curv}(P_{i-1}^{n-1}, P_i^{n-1}) \geq \frac{1}{N_0})$
 then
    proceed Subdivision;
else
    Stop;
end
```

$d(P_{i-1}^{n-1}, P_i^{n-1})$ denotes the euclidean distance between these points, $f_{dist}(d(P_{i-1}^{n-1}, A), d(P_i^{n-1}, A))$ is a function taking into account the euclidean distance $d(P_{i-1}^{n-1}, A)$ and $d(P_{i-1}^{n-1}, A)$, $A \in s(t)$ being the trailing edge point of the air foil and last $g_{curv}(P_{i-1}^{n-1}, P_i^{n-1})$ a function taking into account the curvature of the points.N_0 denotes the user input total number of points on the air foil.

2.1 Distance from trailing edge

the f_{dist} function located in **alt_distance.hpp** as mentioned above takes into account the euclidean distance of the points P_{i-1}^{n-1}, P_i^{n-1} from the trailing edge A the *point_radius* parameter(***Listing 2,Line 9***) adjusts the radius of the effect around the trailing edge and is defined as follows:

$$f_{dist} = \begin{cases} 1, if \ \|P_{i-1}^{n-1}A\| + \|P_i^{n-1}A\| > point\_radius \\ b - (\frac{b-1}{point\_radius})(\|P_{i-1}^{n-1}A\| + \|P_i^{n-1}A\|), \ else \end{cases}$$



(a) point_radius=0.03 (b) point_radius=0.12

Figure 1: Distribution of points after distance from the trailing edge factor for different values of *point_radius* parameter

Listing 2: distance from trailing edge factor

```
double alt_distance_factor(const axlPoint& P1,const axlPoint& P2)
{ axlPoint trailing_edge(1,0,0);

    double distance_P1=axlPoint::distance(P1,trailing_edge);
    double distance_P2=axlPoint::distance(P2,trailing_edge);
    double sum_of_distances=distance_P1+distance_P2;
    double distance_factor;

    double point_radius=0.1;
    double b=10;


    if (sum_of_distances>point_radius)
    {
        distance_factor=1;
    }
    else
    {
        distance_factor=b-((b-1)/point_radius)*sum_of_distances;
    }

    return distance_factor;
}
```

2.2 curvature factor

the g_{curv} function takes into account the curvature of P_{i-1}^{n-1}, P_i^{n-1} , the *region_effect*(***Listing 3,Line 12***) parameter adjusts the sensibility of the effect around the air foil and is defined as follows(***Listing 3,Lines 18-27***):

$$g_{curv} = \begin{cases} b - (\frac{b-1}{region}) \left(\frac{1}{curv(P^{n-1}_{i-1})} + \frac{1}{curv(P^{n}_{i-1})} \right), if \frac{1}{curv(P^{n-1}_{i-1})} + \frac{1}{curv(P^{n}_{i-1})} > region\_effect \\ 1, else \end{cases}$$

where

$$curv(P(u(t), v(t))) = \frac{u(t)'v(t)'' - v(t)'u(t)''}{(u(t)'^2 + v(t)'^2)^{3/2}}$$

denotes the curvature of the point .

Listing 3: curvature factor

```
double curvature_factor(axlAbstractCurveBSpline* axlcurve,double u_a,double u_b)
{
    double curv_a=curvature( axlcurve ,u_a);
    double curv_b=curvature(axlcurve,u_b);
    double radius_a=1/curv_a;
    double radius_b=1/curv_b;

    double sum_of_curvatures=radius_a+radius_b;


    double b=10;
    double region_effect=0.2;
    double curve_factor;




    if (sum_of_curvatures<region_effect)
    {
        curve_factor=(b-((b-1)/region_effect)*sum_of_curvatures);
    }
    else
    {
        curve_factor=1;
    }

    return curve_factor;

}
```

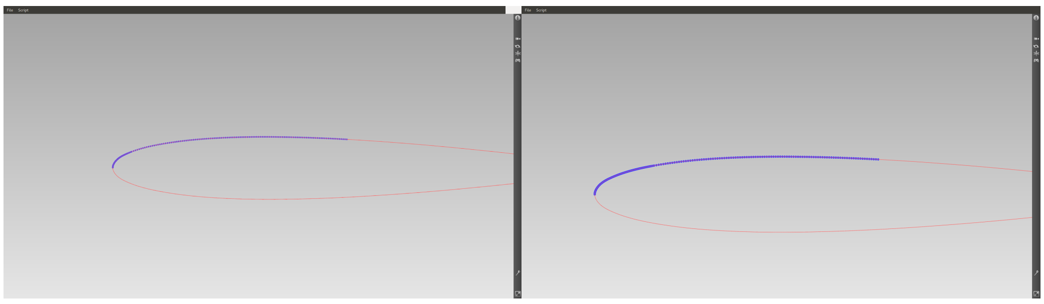

(a) curvature_effect=0.2 (b) curvature_effect=0.7

Figure 2: Distribution of points after distance from the trailing edge factor for different values of *region_effect* parameter

2.3 Distribution of points selected for the construction of the physical mesh

Now that we assembled $P_i \in s(t)$,$i \in \{0, 1, ...N\}$ we proceed by constructing the physical mesh of points. By collecting $X_i \in k(\tau)$ in uniform fashion we then choose points Q_{ij} along the line segments P_iX_i .The selection takes place in **get_meshes.hpp** , **get_meshes_contour(...)** function after the subdivision algorithm.
If M is the input of points on line variable of the user then(***Listing 4,25***) :

$$Q_{ij} = P_i + (X_i - P_i) * l_0 * \frac{a^{j+1} - 1}{a - 1}, where\ i \in \{0, 1, ..., N\}\ and\ j \in \{0, 1, ..M\}$$

where $l_0 = \frac{a-1}{a^M - 1}$ and a is a scaling factor.

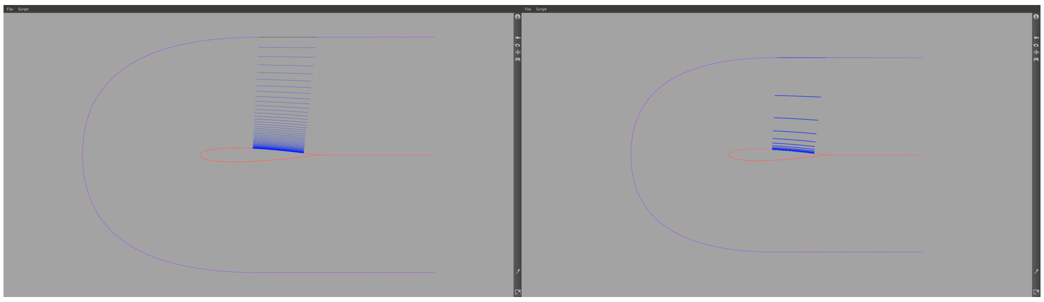

(a) a=1.1 (b) a=1.7

Figure 3: The effect of scaling factor a on selection of points

Listing 4: Creation of the physical mesh of points

```
void get_meshes_contour( axlAbstractCurveBSpline* axlcurve, axlAbstractCurveBSpline*
      contour_curve,axlMesh* physical_mesh,axlMesh* parameter_mesh,const double
      points_on_line, double points_on_curve)
{...
   //initialization of selection of points v−direction for each of the subdivision points
    step=0;
    std::vector<double> step_vector;
    for(int i=0;i<=points_on_line;i++)


    {
          step_vector.push_back(step);
    }

    for (int i=0;i<points_on_curve;i++)
    {
          axlPoint Pi=P.at(i);
          axlPoint X=contour_curve−>eval(step);
          physical_mesh−>push_back_vertex(P.at(i));

          double distance_PiX=axlPoint::distance(Pi,X);
          distances_d.push_back(dst.distance(Pi.x(),Pi.y(),Pi.z()));
          distances_l.push_back(0);

          for(double j=1;j<=points_on_line;n++)

          { axlPoint Q=Pi+(X−Pi)*(l0*(((pow(a,n+1))−1)/(a−1)));
                physical_mesh−>push_back_vertex(Q);


                double distance_QPi=axlPoint::distance(Q,Pi);
                double distance_l=distance_QPi/distance_PiX;
                double distance_d=dst.distance(Q.x(),Q.y(),Q.z());

                if (j==points_on_line)
                {
                      distances_l.push_back(1);
                }
                else
                {

                      distances_l.push_back(distance_l);}
                distances_d.push_back(distance_d);
          }

          step+=1/(points_on_curve);

          for(int i=0;i<=points_on_line;i++)
```

```
        {
            step_vector.push_back(step);
        }

... }
```

2.4 Parametric Domain

In **get_meshes.hpp**,**get_meshes_contour(...)** we then proceed by constructing the parameter domain $\widehat{\Omega}$ which will be mapped to the physical domain Ω via the fitting process.Coordinates (u_i, v_i) are equal to:

$$u_i = i/N$$

where $i \in \{0, 1, 2, ..M\}$,M being the number on the points on curve determined the subdivision algorithm.and

$$v_i = log(v_{i_0} + b_1) - \frac{log(b_1)}{log(a_1)}$$

which is a result of the logarithmic transformation of(***Listing 5,Line 21***):

$$v_{i_0} = \delta\|Q_{ij}P_i\| + (1-\delta)(\frac{\|Q_{ij}P_i\|}{\|P_iX_i\|})$$

and

$$\delta = \begin{cases} 0, if\|Q_{ij}P_i\| < d_0 \\ 2d_0 - \|Q_{ij}P_i\|, if\|Q_{ij}P_i\| < 2d_0 \\ 0, else \end{cases}$$

where $a_1 = a^M - 1$ and $b_1 = \frac{1}{a-1}$ and M is total of points along the line segments P_iX_i.Thus taking into account the barycentric position of Q_{ij} along the line segment P_iX_i when we're closer to the air foil(d_0 affecting the criteria).

With the parametric and physical mesh in place we now proceed to the fitting process located in **get_surface.hpp**. The parametric coordinates are then mapped to the physical domain via the function $S(u, v) = \sum_i \sum_j R_{ij}(u, v)d_{ij}$,where d_{ij} are the control points determined by the fitting algorithm(***Listing 6,Line 26***).

Listing 5: Parametric coordinates

```
void get_meshes_contour( axlAbstractCurveBSpline* axlcurve, axlAbstractCurveBSpline*
        contour_curve,axlMesh* physical_mesh,axlMesh* parameter_mesh,const double
        points_on_line, double points_on_curve)
{...

//Creating parameter mesh
    double delta;
    std::vector <double> v;
    for(int j=0;j<distances_d.size();j++) {
        if(distances_d.at(j)<=d0)
        {
            delta=1;
        }
        else if( (distances_d.at(j)<=2*d0) )
        {
            delta=(((2*d0)-distances_d.at(j)));

        }else
        {
            delta=0;
        }
        double v0=((delta)*distances_d.at(j))+((1-delta)*distances_l.at(j));
        v0=(log(v0+b1)-log(b1))/log(a1);
        v.push_back(v0);

        parameter_mesh->push_back_vertex(step_vector.at(j),v.at(j),0);

    }

  ... }
```

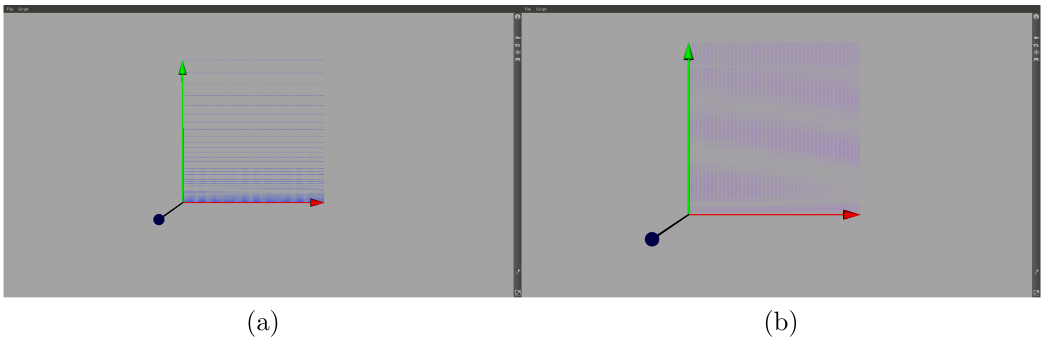

(a) (b)

Figure 4: Parametric mesh before(a) and after(b) logarithmic transformation

Listing 6: fitting process

```
#ifndef GET_SURFACE_HPP
#define GET_SURFACE_HPP
#include <axlCore/axlAbstractSurfaceBSpline.h>
#include <axlCore/axlMesh.h>


axlAbstractSurfaceBSpline* get_surface(axlMesh* physical_mesh, axlMesh* parameter_mesh)

{ int rfnmt_loops=4;


    BsplineBasis2D basis_for_mesh(60,40);
    static SplineSurface *fitted_surface=new SplineSurface;
    double* err_new=new double;
    *err_new=1.e-9;
    SplineSurface *presurf=new SplineSurface;
    presurf=NULL;
    int error_method=0;//SDM
    int opt_method=0; //EDGES FIT=1|GLOBAL_OPT=0
    double lambda=1.e-2;

    RowVectorXd* curve_vector=new RowVectorXd;

    RowVectorXd* error_vector=new RowVectorXd(physical_mesh->vertex_count());
    error_vector->setZero();
    fitted_surface=fitting<axlMesh,axlMesh,BsplineBasis2D>::compute_fit_surf(
        physical_mesh,parameter_mesh,basis_for_mesh,presurf,error_method,opt_method,
        lambda,err_new,error_vector,curve_vector,rfnmt_loops);

    axlAbstractSurfaceBSpline* output=new axlAbstractSurfaceBSpline;
    output=dynamic_cast<axlAbstractSurfaceBSpline*>(dtkAbstractDataFactory::instance
        ()->create("goSurfaceBSpline"));
    SplineSurface* tmp = static_cast<SplineSurface *>(output->surface());
    *tmp = *fitted_surface;
    return output;
}

#endif // GET_SURFACE_HPP
```

3 Generating surface between bodies

Let $d(\tau)$ represent the curve describing a different air foil curve.The same properties described above between the two air foil bodies must apply as well.Using the subdivision described above we proceed by introducing a new subdivision algorithm(**get_meshes.hpp**,**get_meshes_body(...)**) this time taking place along the line segments P_iX_i,$P_i \in s(t)$,$X_i \in d(\tau)$ which calculates new points(***Listing 7,Line 39***):

$$Q_{ij}^n = \frac{Q_{ij-1}^{n-1} + Q_{ij}^{n-1}}{2}$$

where $Q_{ij-1}^{n-1}, Q_{ij}^{n-1} \in P_iX_i$, $Q_{ij-1}^0 \equiv P_i$ and $Q_{ij}^0 \equiv X_i$.

If $(u_i, v_i) \in \widehat{\Omega}$ represent the parametric coordinates of the air foil then the new subdivision algorithm functions under the criteria(***Listing 7,Line 37***):

```
if $|u_i^{n-1} - u_{i-1}^{n-1}| \geq \frac{1}{M_0}$ then
    proceed Subdivision;
else
    Stop;
end
```

Listing 7: Subdivision Algorithm

```
1  void get_meshes_body( axlAbstractCurveBSpline* axlcurve, axlAbstractCurveBSpline* contour_curve,axlMesh* physical_mesh,axlMesh* parameter_mesh, double points_on_line , double points_on_curve)
2  {...
3  for (int i=0;i<P.size();i++)
4
5      {
6          std::vector<double> v;
7          std::vector<axlPoint> Q;
8          axlPoint Q_start=P.at(i);
9          Q.push_back(Q_start);
10         u.push_back(0);
11         axlPoint Q_finish=contour_curve->eval(step);
12         Q.push_back(Q_finish);
13         u.push_back(1);
14
15
16         std::map<int,int> edge;
17         edge[0]=1;
18         int b,e;
19         std::list<int> stack;
20         stack.push_front(0);
```

```
        count=2;
        distfield_curve<axlAbstractCurveBSpline>* dst=new distfield_curve<axlAbstractCurveBSpline> (axlcurve,100000);
        while(!stack.empty())
        {

            std::list<int>::iterator it=stack.begin();
            b=*it;
            stack.pop_front();
            e=edge[b];
            axlPoint Q_b=Q.at(b);
            axlPoint Q_e=Q.at(e);
            double v_b=v.at(b);
            double v_e=v.at(e);



            if(fabs(v_b-v_e)>1/points_on_line)
            {
                axlPoint Q_middle_on_line=(Q.at(b)+Q.at(e))/2;
                double v_middle=v_value(Q_middle_on_line,Q_start,Q_finish,dst,points_on_line);
                v.push_back(v_middle);
                Q.push_back(Q_middle_on_line);


                int n=Q.size()-1;
                edge[b]=n;
                edge[n]=e;
                stack.push_back(b);
                stack.push_back(n);

                count++;

            }

    ... }
```

This times we are calculating the parameter domain coordinate v_i which serves as well as the stopping criteria for subdivision. Let

$$v_{i_0} = (1-\theta)\|\frac{Q_{ij}^n P_i}{Q_{ij}^n X_i}\| + \theta Q_{ij}^n P_i$$

where:

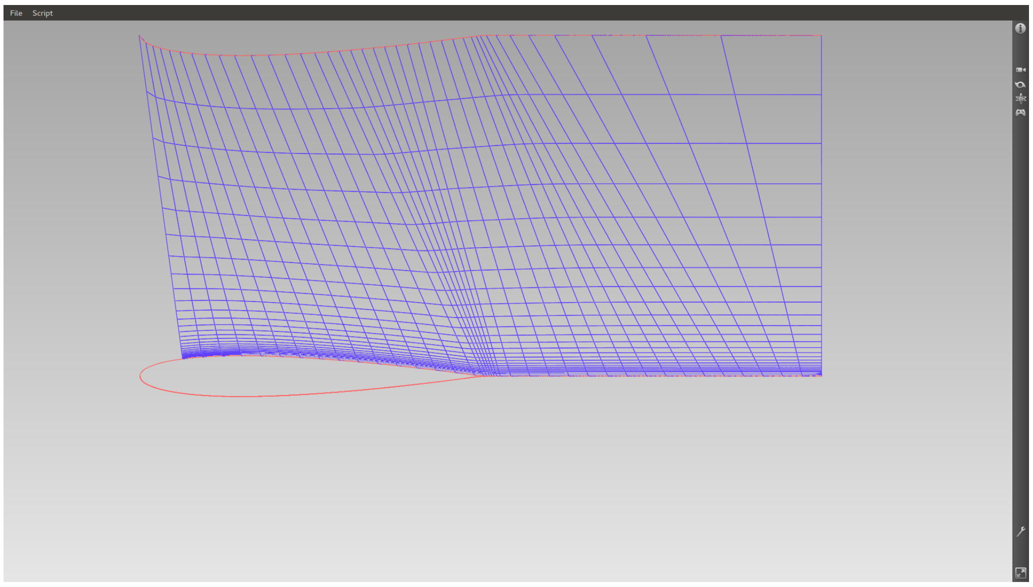


Figure 5: Surface generated between two air foils

$$\theta = \begin{cases} 1, if\ \|Q_{ij}^n P_i\| < d_0 \\ 2d_0 - \frac{\|Q_{ij}^n P_i\|}{d_0}, if\ \|Q_{ij}^n P_i\| < 2d_0 \\ 0, else \end{cases}$$

The points v_{i_o}remain the same or submitted to a logarithmic transform depending on d_0 thus obtaining the parametric coordinates v_i(***Listing 7,Lines 30-39***) ,while $u_i = i/N$ as in the previous case.

Listing 8: Subdivision Criteria

```
double v_value(const axlPoint &Q,const axlPoint &P,const axlPoint &X,distfield_curve<
    axlAbstractCurveBSpline>* dst,const double &points_on_line)
{
    double r=1.1;
    double a=pow(r,points_on_line-1);
    double d0=1;

    double b=d0/(pow(a,d0)-1);



    double distance_Q=dst->distance(Q.x(),Q.y(),Q.z());
    double barycentric_Q=axlPoint::distance(Q,P)/axlPoint::distance(P,X);
    double theta,v0,v;

    if(distance_Q<d0)
    {
        theta=1;
    }
    else if(distance_Q>2*d0)
    {
        theta=0;
    }
    else
    {
        theta=2-(distance_Q/d0);
    }
    theta=0;
    v0=(1-theta)*barycentric_Q+distance_Q*theta;

    if (v0<=d0)
    {
        v=(log(v0+b)-log(b))/log(a);
    }
    else
    {
        v=v0;

    }
    return v;

}

}
```