In general, the equations are solved as follows:

- Start with the first and last equations, and work inward, computing the new control points.
- If the number of control points is even (e.g., equ's B), then the final new control point is computed twice. The knot can be removed if the two values of the control point are within tolerance.

- If the number of equations is odd (e.g., equ's C), then all new control points are computed once, and the last two computed are substituted into the middle equation. If the result is in tolerance of the old control point on the left-hand side of the middle equation, then the knot is removable.

The following formulas generalize what we have observed from the simple example.

Let $u = u_r$ be a knot of multiplicity s, where $1 \le s \le p$ (we assume $u \ne u_{r+1}$). The equations for computing the new control points for one knot removal of u are:

$$\boldsymbol{P}_i^1 = \frac{\boldsymbol{P}_i^0 - (1 - \alpha_i)\boldsymbol{P}_{i-1}^1}{\alpha_i}$$

$$, \quad r - p \le i \le \frac{2r - p - s - 1}{2}$$

$$\boldsymbol{P}_j^1 = \frac{\boldsymbol{P}_j^0 - \alpha_j \boldsymbol{P}_{j+1}^1}{(1-\alpha_j)}$$

$$, \quad \frac{2r-p-s+2}{2} \le j \le r-s$$

$$with$$

$$\alpha_k = \frac{u-u_k}{u_{k+p+1}-u_k} \qquad , \quad k = i,j$$

A simple program to accomplish this is,

```
$i = r - p; \quad j = r - s;$
$while\,(j - i > 0)$
    $\alpha_i = \ldots$
    $\alpha_j = \ldots$
    $\boldsymbol{P}_i^1 =$
    $\boldsymbol{P}_j^1 =$
    $i = i + 1; \quad j = j - 1;$
```

Now suppose we want to remove $u = u_r$ multiple times. Each time the knot is removed, s and r are decremented, and the superscripts on the control points are incremented. Thus the equations for removing $u = u_r$ the t-th time are,

$$\boldsymbol{P}_i^t = \frac{\boldsymbol{P}_i^{t-1} - (1-\alpha_i)\boldsymbol{P}_{i-1}^t}{\alpha_i}$$

$$, r - p - t + 1 \le i \le \frac{2r - p - s - t}{2}$$

$$\boldsymbol{P}_j^t = \frac{\boldsymbol{P}_j^{t-1} - \alpha_j \boldsymbol{P}_{j+1}^t}{(1-\alpha_j)}$$

$$, \quad \frac{2r-p-s+t+1}{2} \le j \le r-s+t-1$$

$$with$$

$$\alpha_i = \frac{u-u_i}{u_{i+p+t}-u_i} \quad and \quad \alpha_j = \frac{u-u_{j-t+1}}{u_{j+p+1}-u_{j-t+1}}$$

Assuming *u* is to be removed *k* times, this equation can be programmed as follows:

```
$first = r - p + 1;$   $last = r - s - 1;$
        $for\ t = 1, \ldots, k$
  $first = first - 1;$          $last = last + 1;$
      $i = first;$          $j = last;$
         $while\ (j - i > t - 1)$
             $\alpha_i = \ldots$
             $\alpha_j = \ldots$
```

$$\boldsymbol{P}_i^1 =$$

$$\boldsymbol{P}_j^1 =$$

$$i = i + 1; \quad j = j - 1;$$

The factor which complicates the implementation of knot removal is that it is generally unknown in advance whether a knot is removable, and if it is, how many times.

The (potentially) new control points must be computed in temporary storage, and it is not known in advance how many knots and control points will be in the output.

The following algorithm takes this approach. It tries to remove the knot $u = u_r$ *num* times, where $1 \leq num \leq s$. It returns t, the actual number of times the knot was removed, and if $t > 0$, it returns the new knot vector and control points.

It computes the new control points in place, overwriting the old ones. Only one local array (**`temp`**) of size $2p + 1$ is required to compute the new control points at each step. If removal fails at step t, the control points from step $t - 1$ are still intact.

At the end, knots and control points are shifted down to fill the gap left by removal.

To check for coincident points, a value **TOL** is assumed, and a function **Distance4D()**, which computes the distance between points. If the curve is rational, **Distance4D()** computes the 4D distance (treating the weights as ordinary coordinates). The parameter **TOL** has the following meaning:

- if the curve is nonrational (all $w_i = 1$), then one knot removal results in a curve whose deviation from the original curve is less than **TOL**, the entire parameter domain.

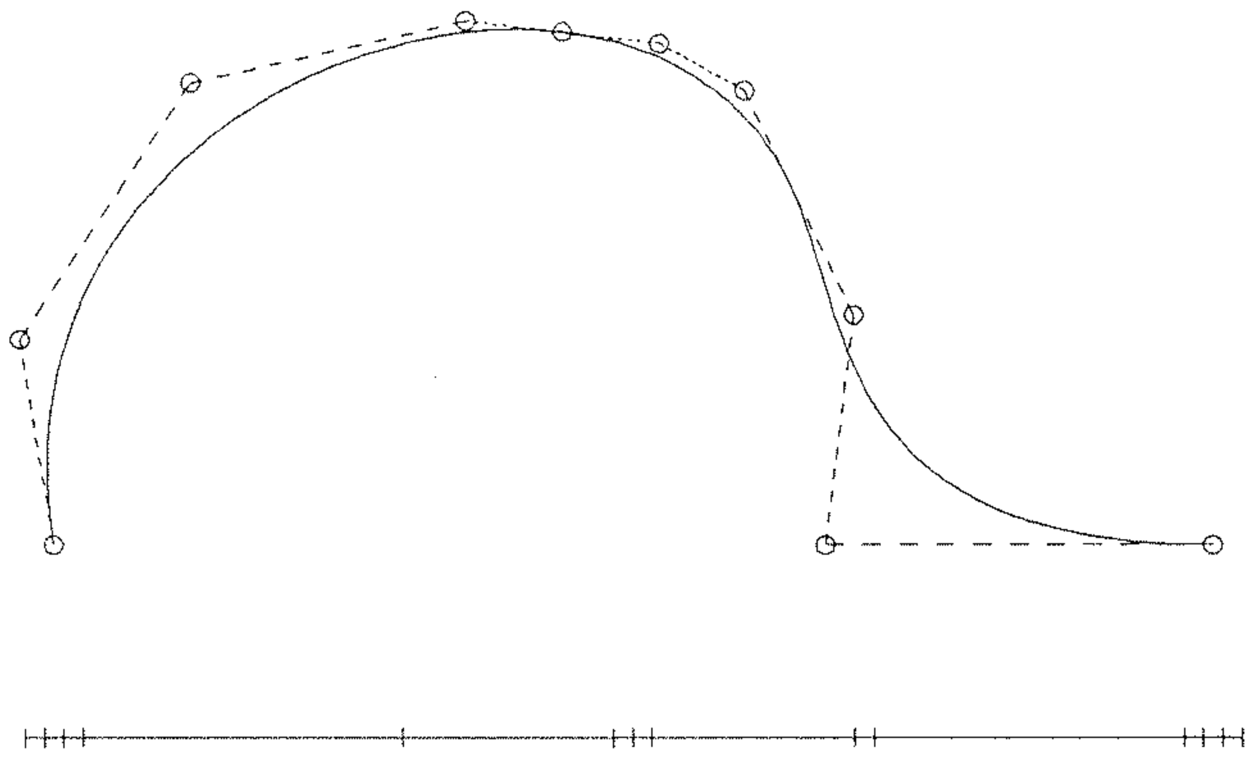
- if the curve is rational, then the deviation is everywhere less than

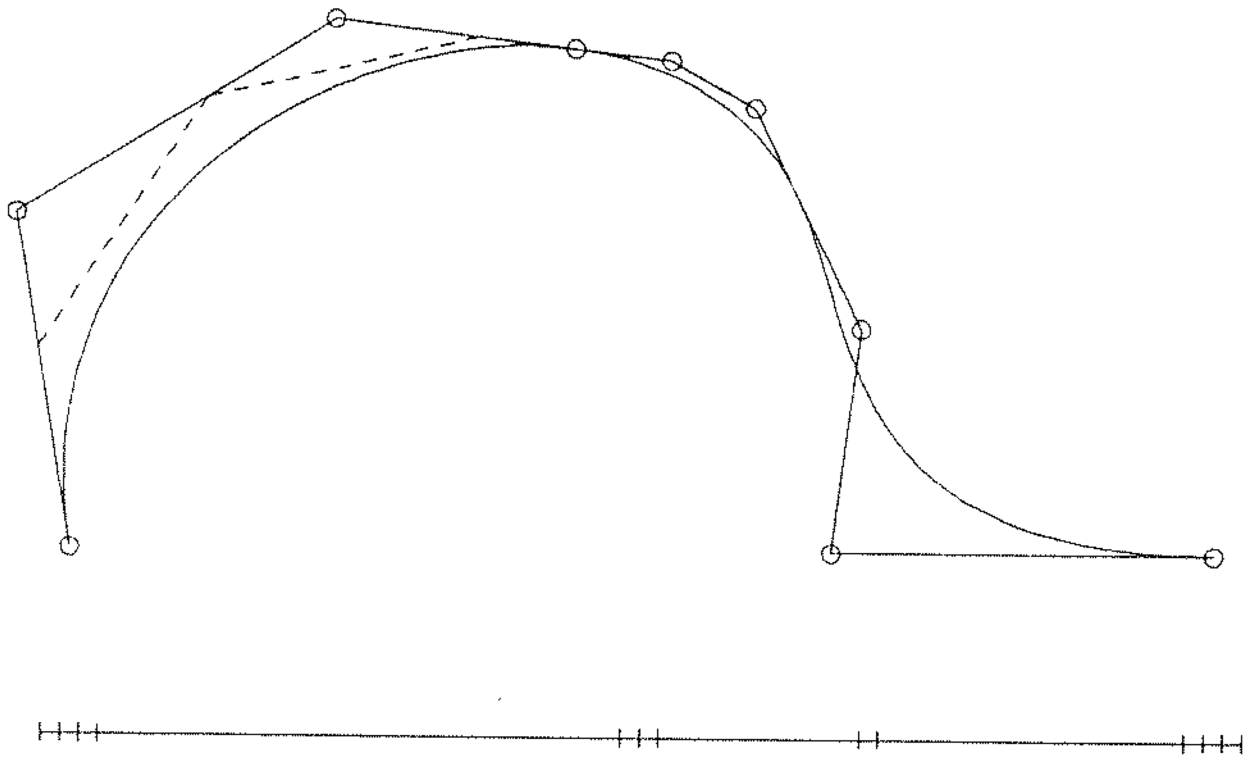
$$\frac{\mathrm{TOL} \cdot (1 + |\boldsymbol{P}|_{max})}{w_{min}}$$

where w_{min} is the minimum weight on the original curve, and $|\boldsymbol{P}|_{max}$ is the maximum distance of any (3D) point on the original curve from the origin.

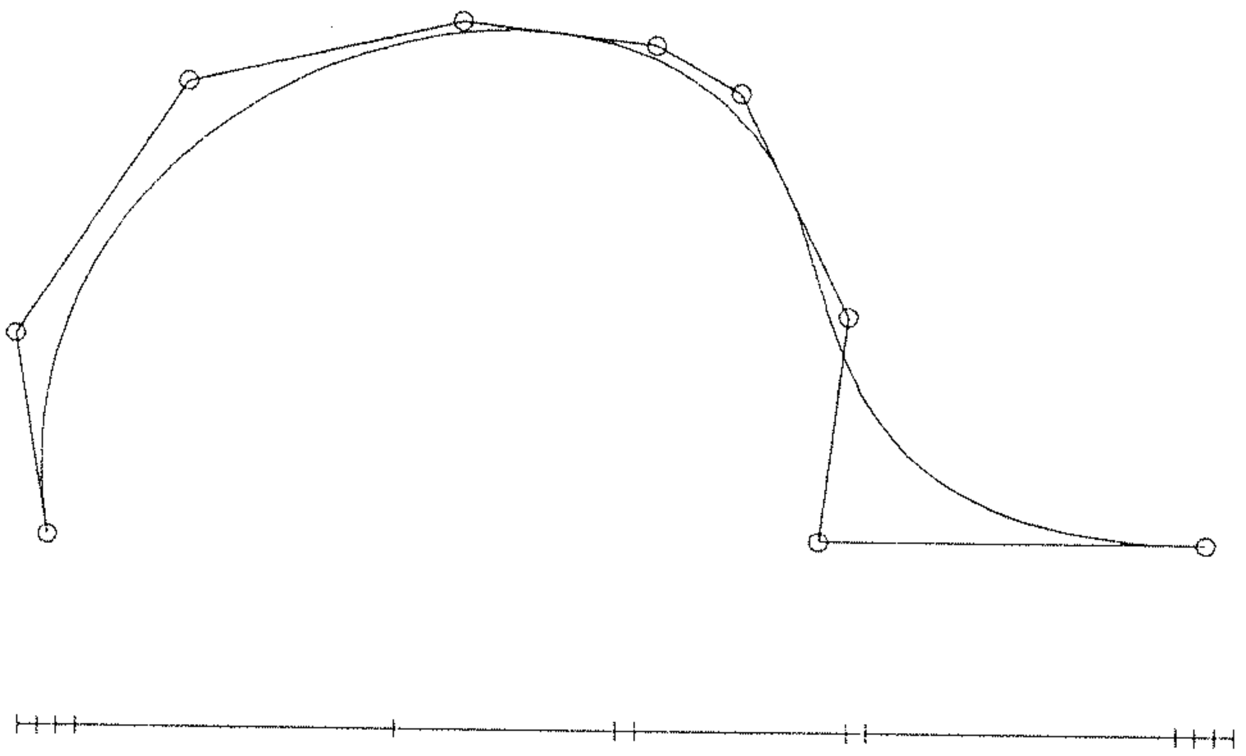
The convex hull property of B-splines can be used to compute bounds for w_{min} and $|\boldsymbol{P}|_{max}$. If the desired bound on deviation is d, then **TOL** should be set to:

$$\text{TOL} = \frac{d \cdot w_{min}}{1 + |\boldsymbol{P}|_{max}}$$

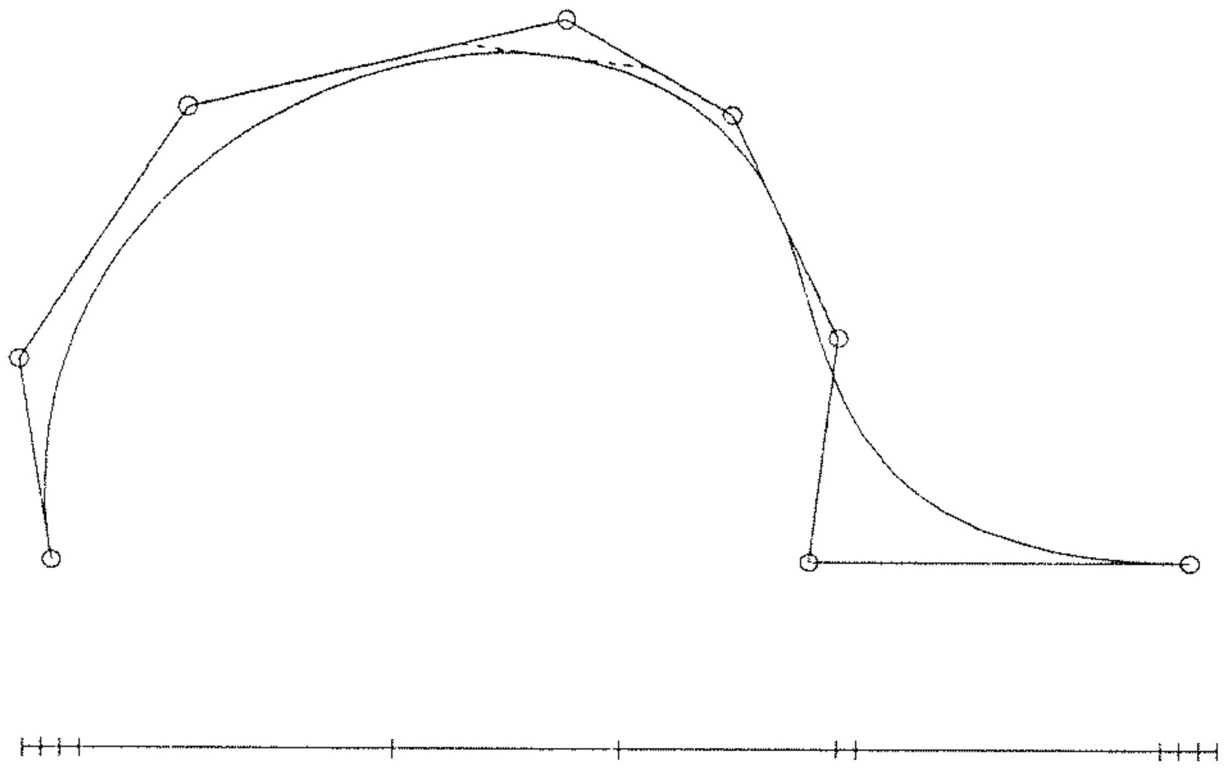

original curve



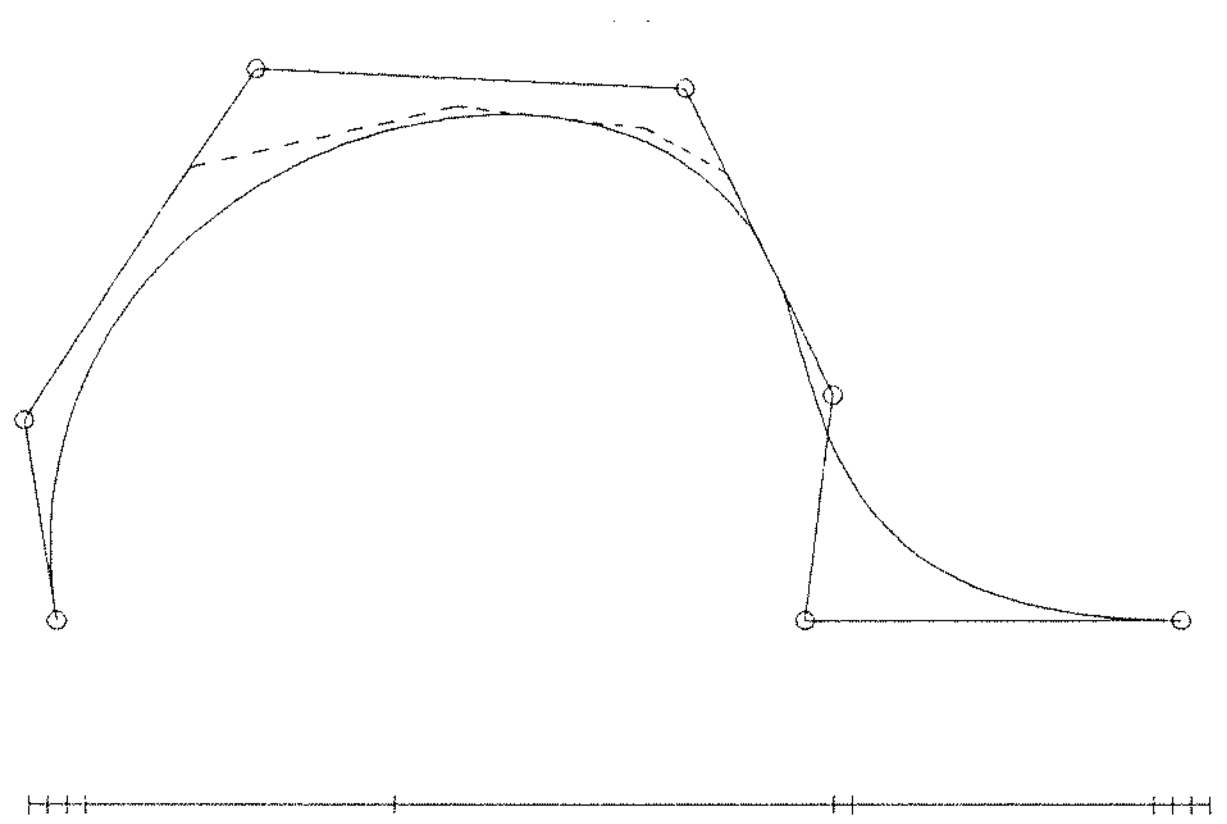
u=0.3 removed



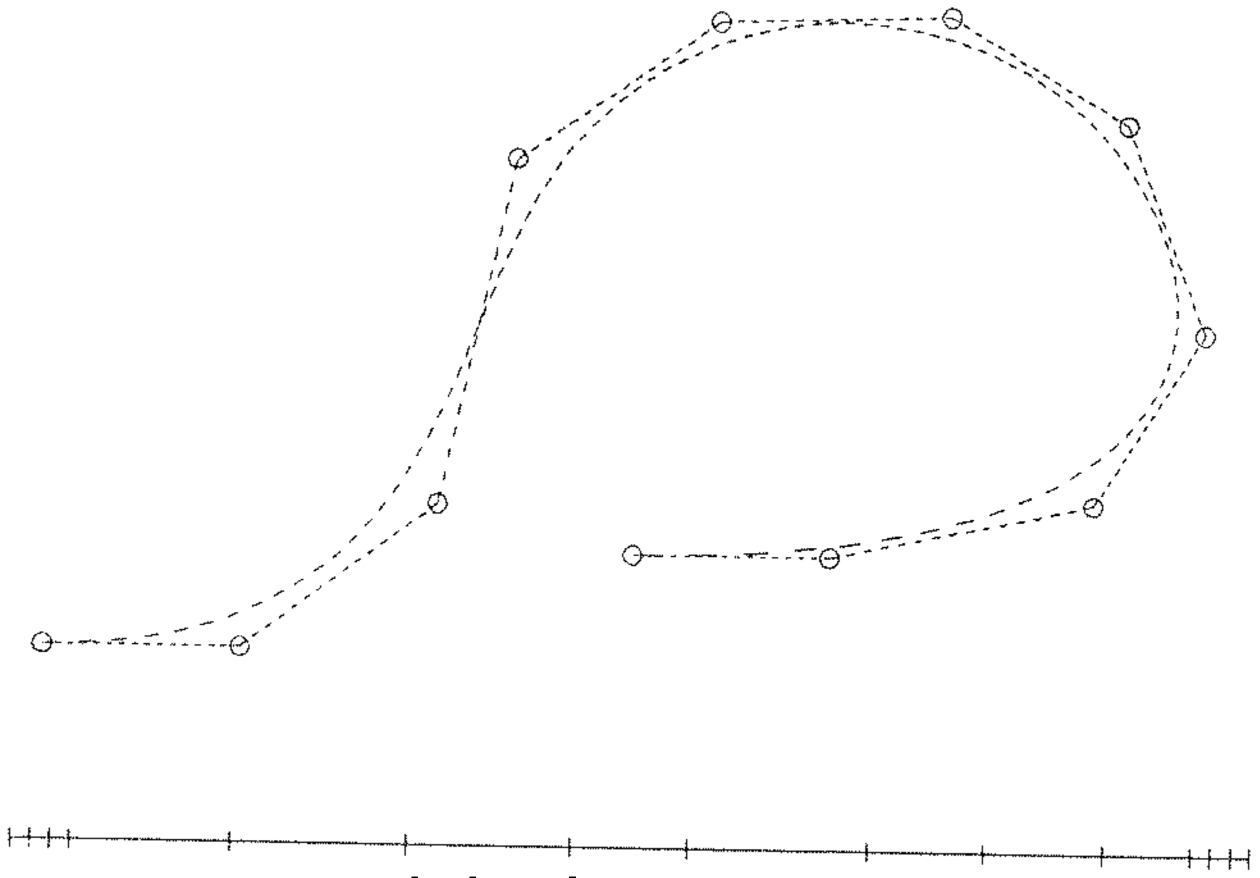
u=0.5 removed once

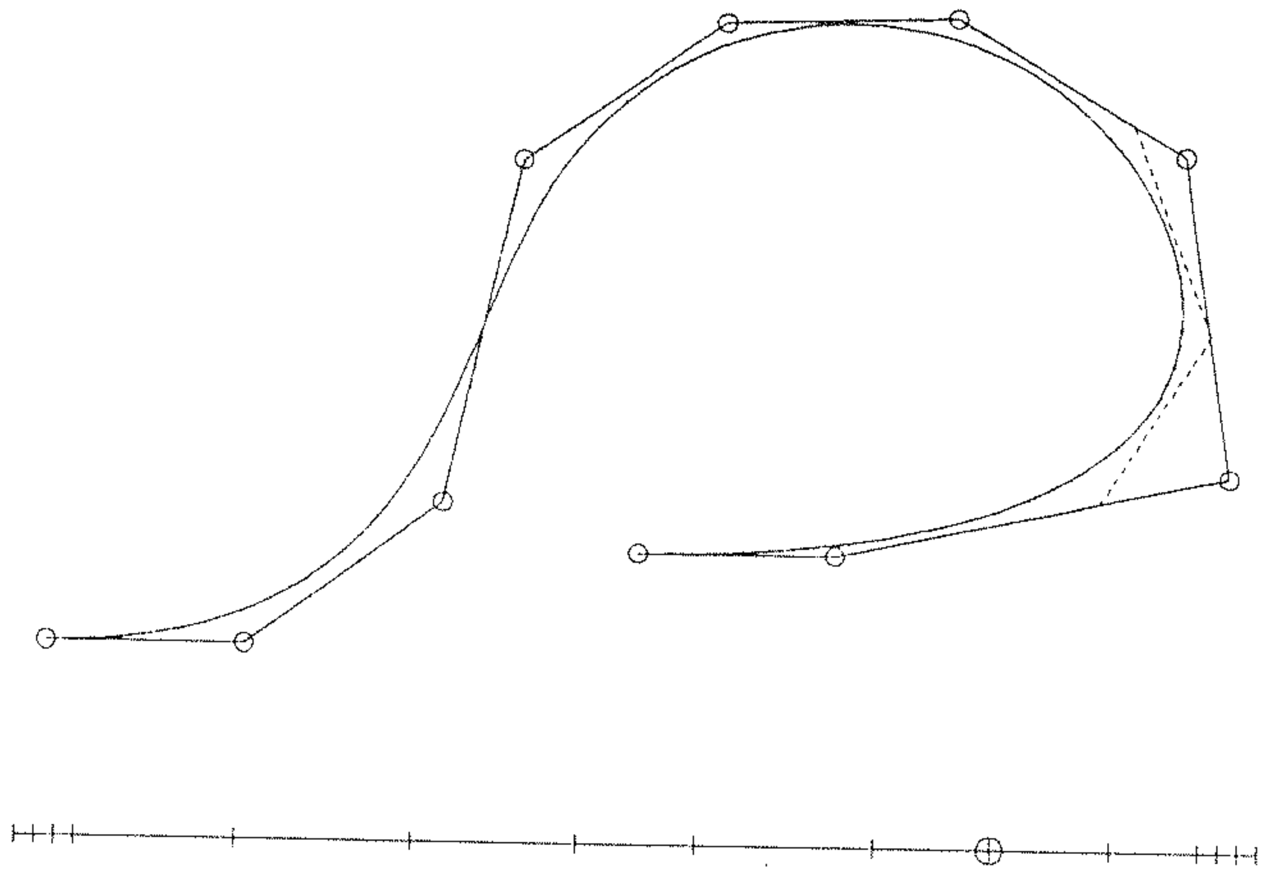


u=0.5 removed twice

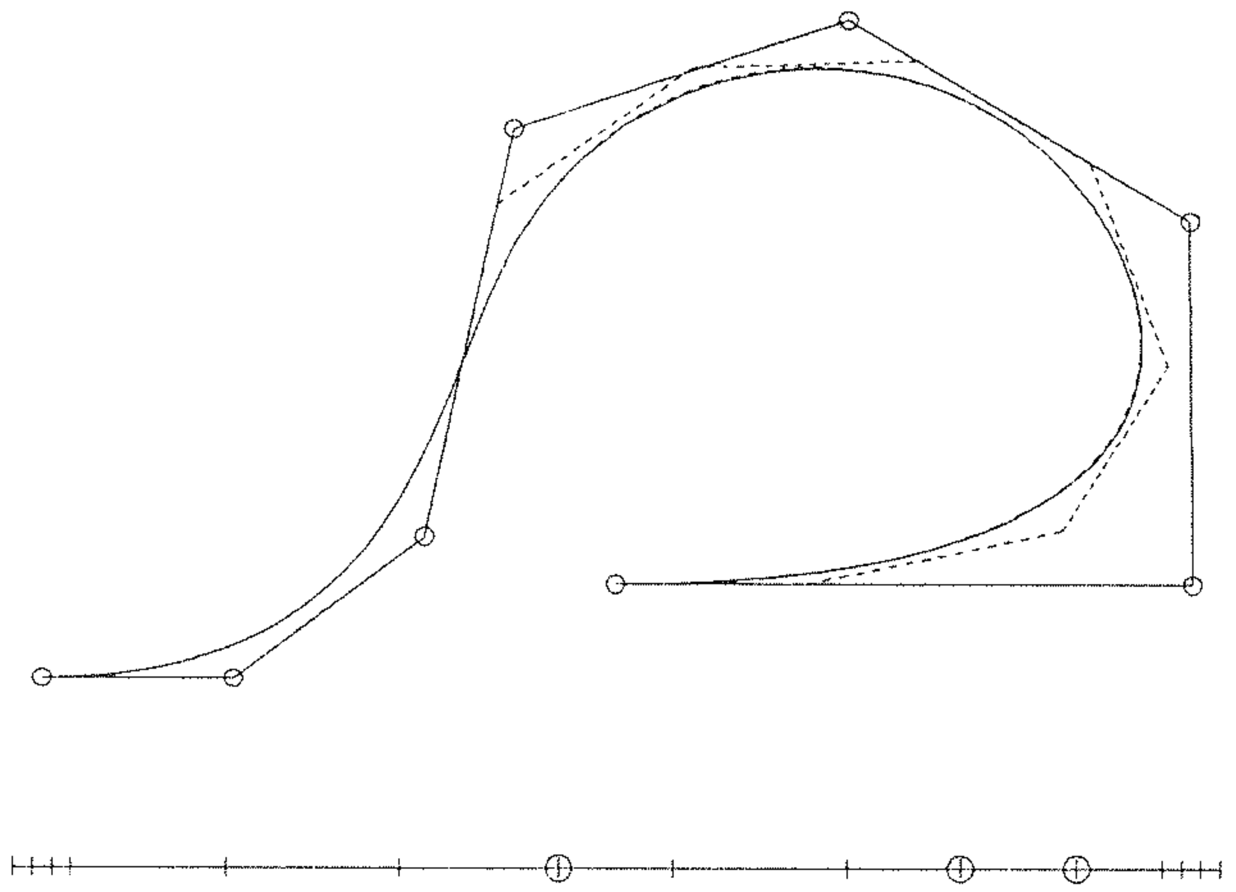


u=0.5 remove 3rd time

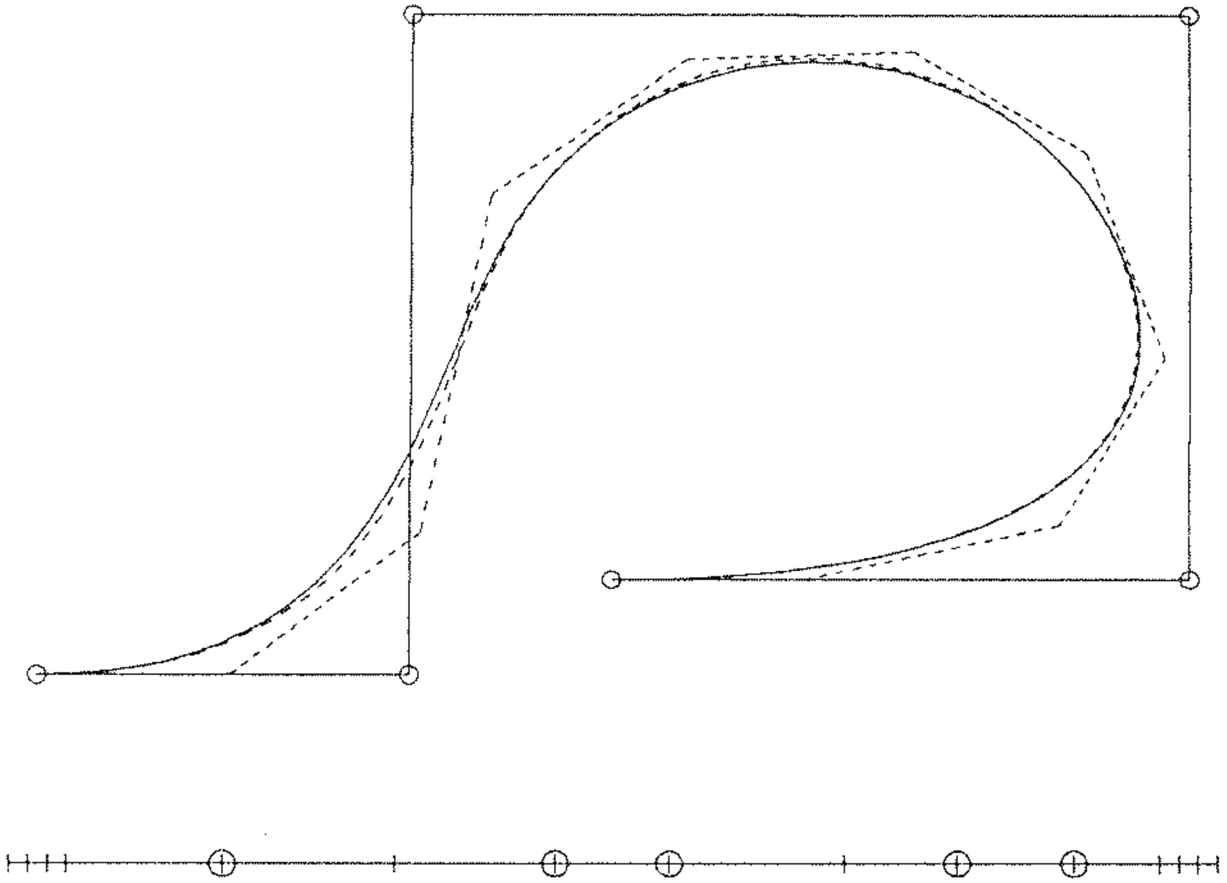

original curve



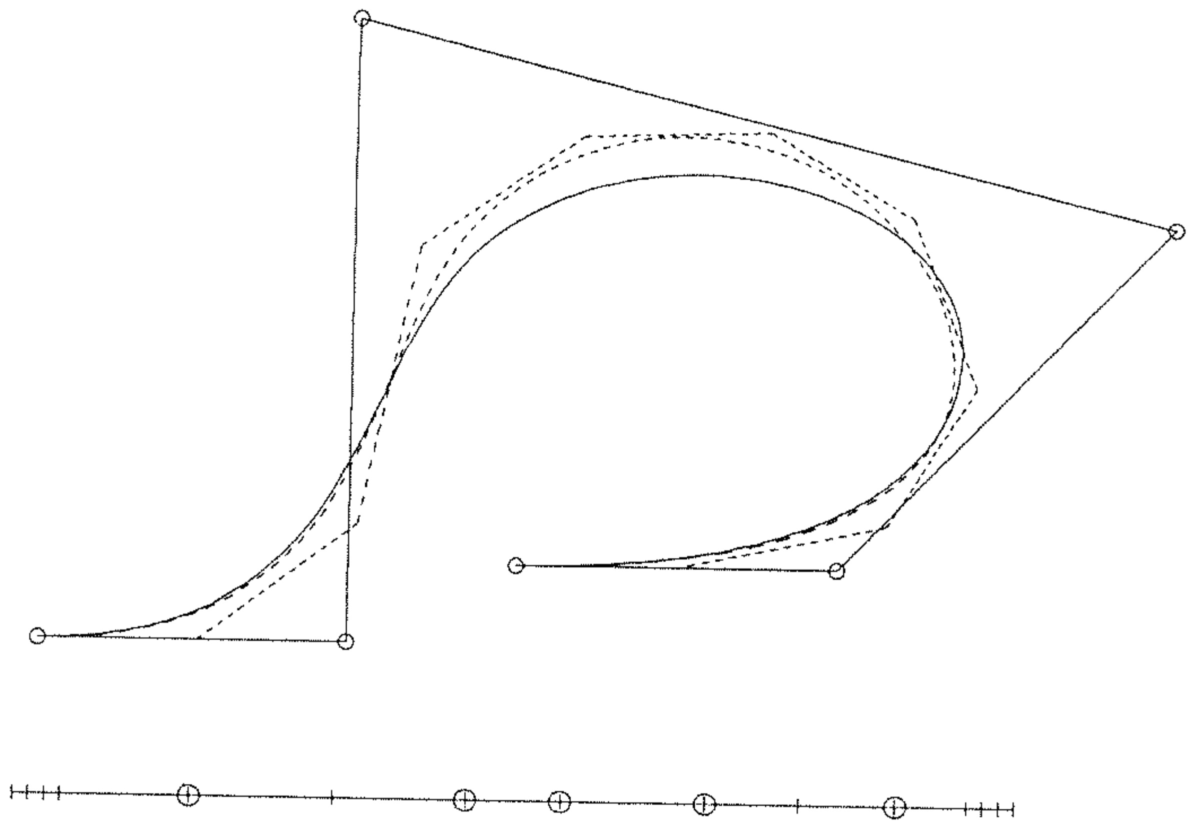
TOL=0.007



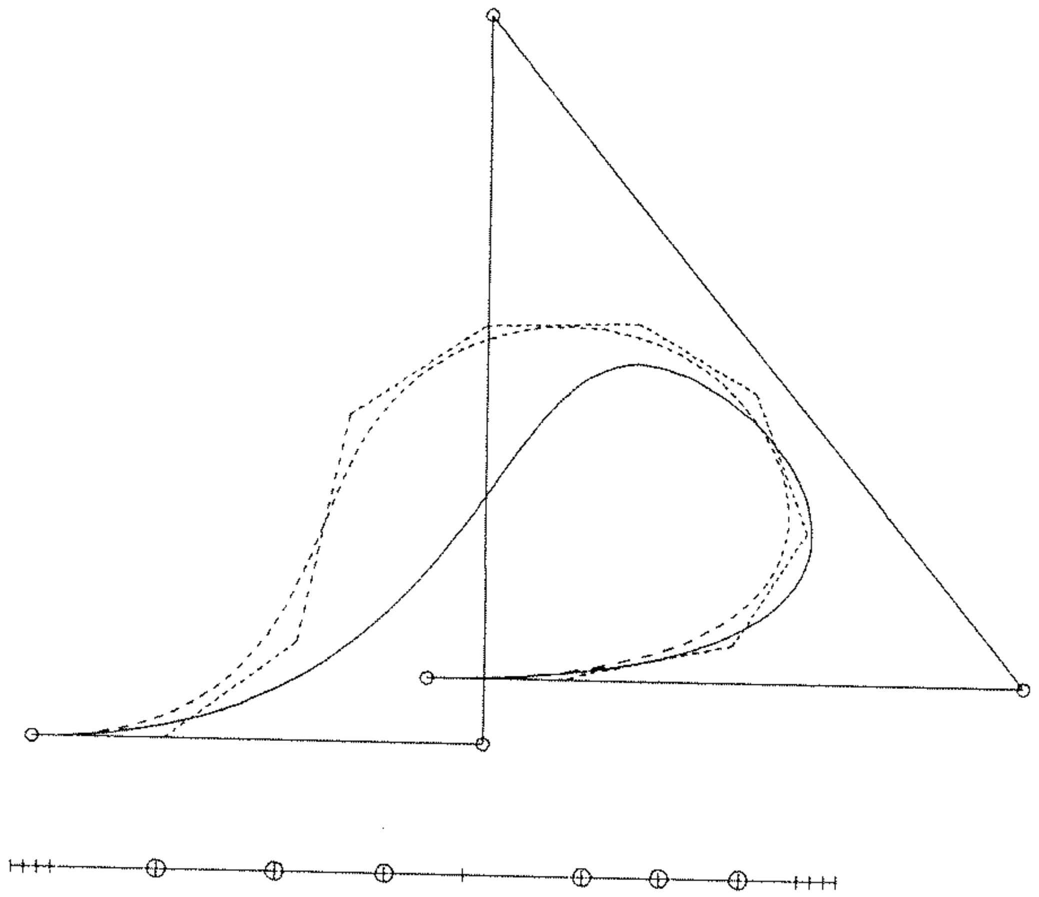
TOL=0.025



TOL=0.07



TOL=0.6



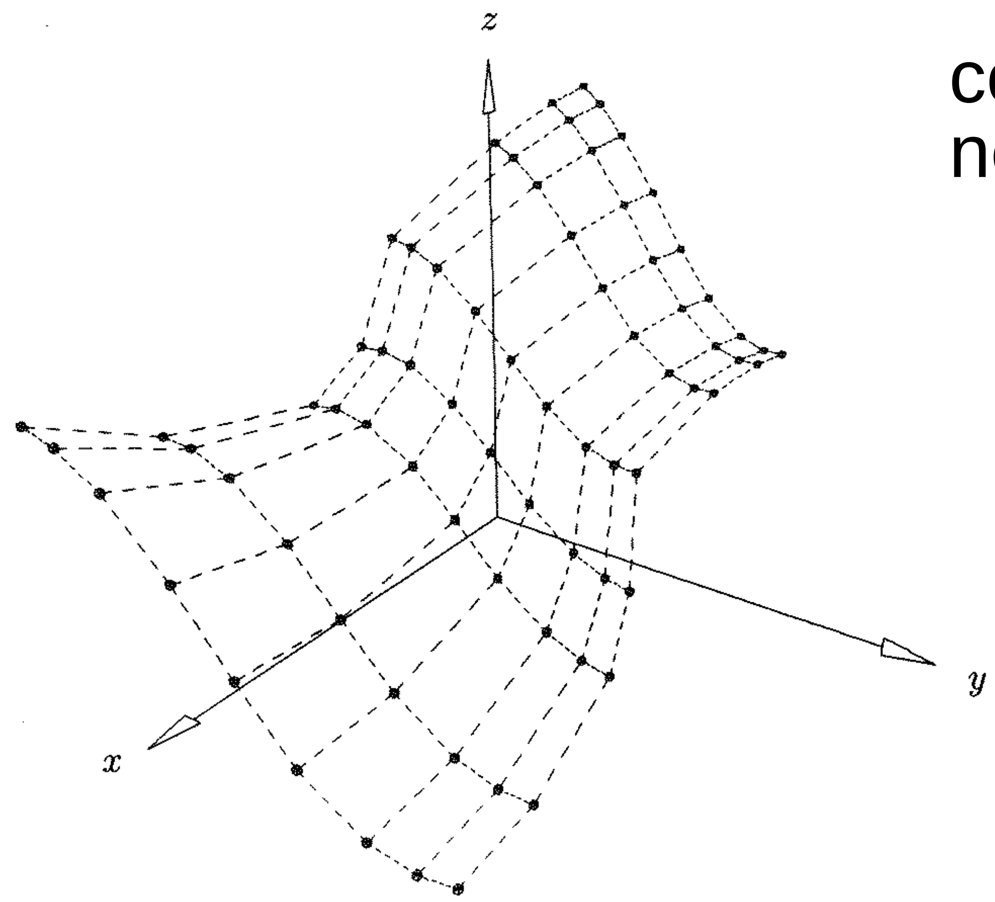
TOL=1.2

Note that Algorithm A5.8 may create negative or zero weights. Theoretically, this is correct, but it may be undesirable for software reasons. It can be avoided by simply inserting a check for this after the distance computation, before setting **`remflag`** to 1.

For a NURBS surface

$$\boldsymbol{S}^w(u,v) = \sum_{i=0}^{n}\sum_{j=0}^{m} N_{i,p}(u)\,N_{j,q}(v)\,w_{ij}\boldsymbol{P}_{ij}^w$$

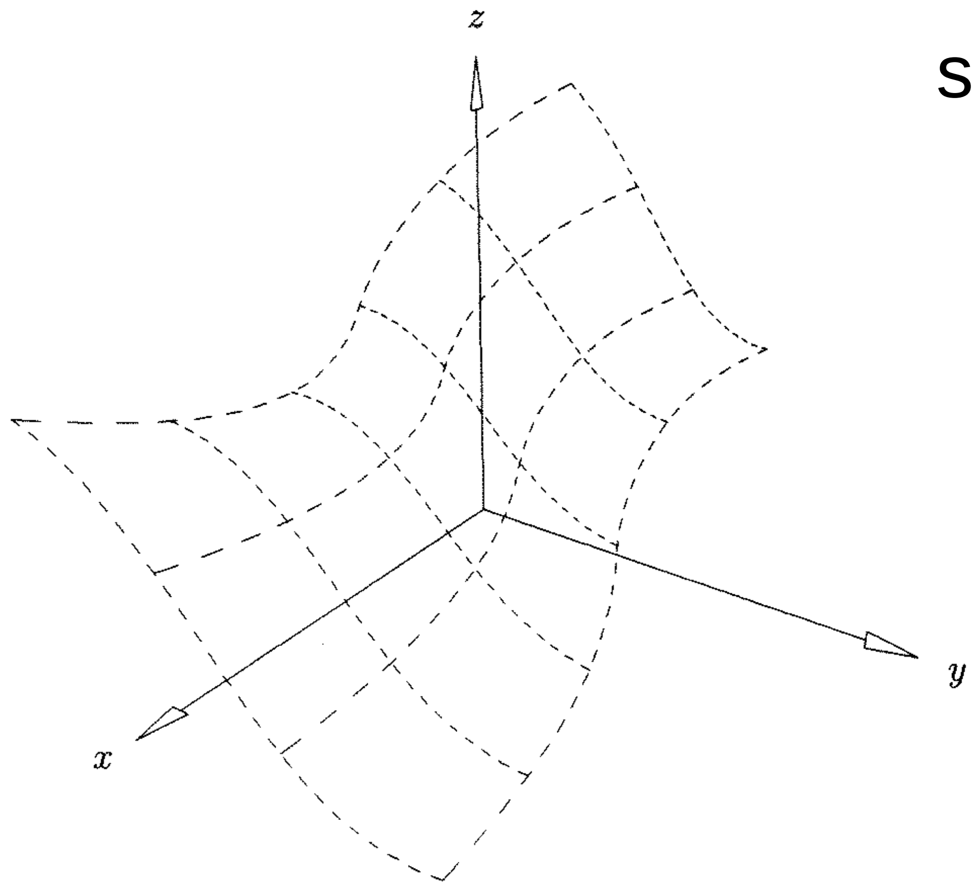
a *u*-knot (*v*-knot) is removed by applying the above knot removal algorithm to the *m* + 1 columns (*n* + 1 rows) of the control points. But the knot may be removed only if the removal is successful for all *m* + 1 columns (*n* + 1 rows).



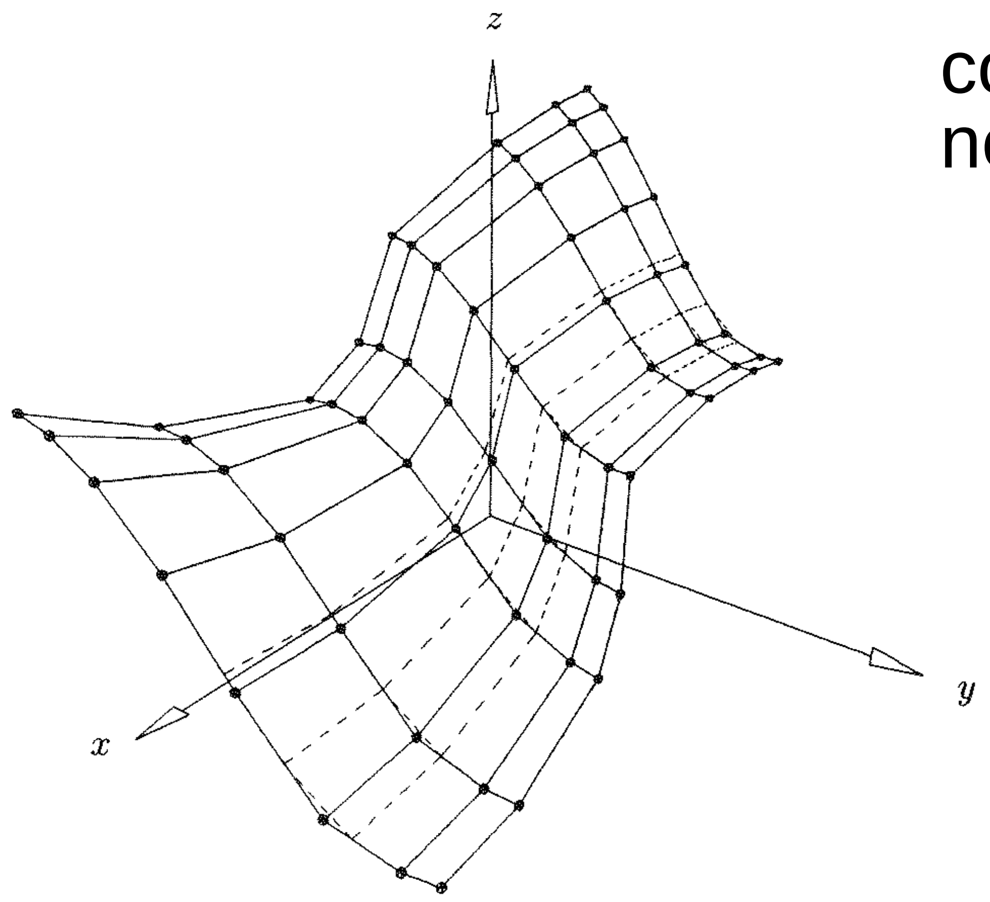
control
net

original surface

surface



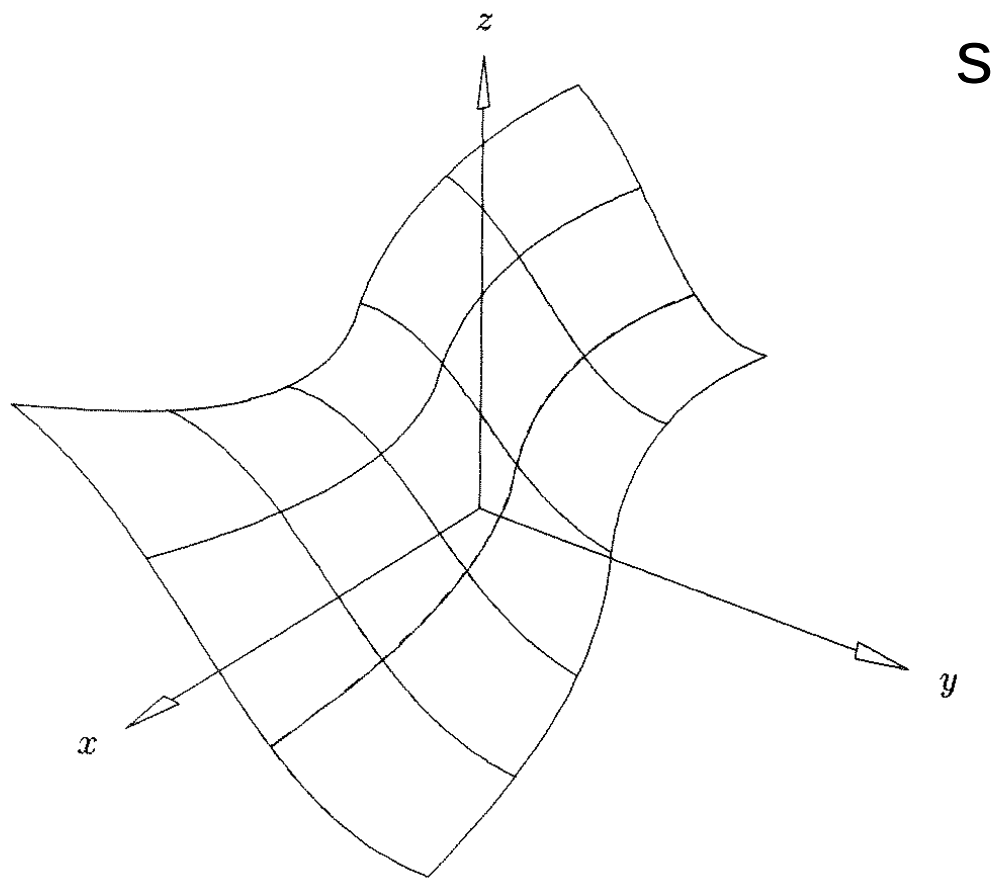
original surface



control
net

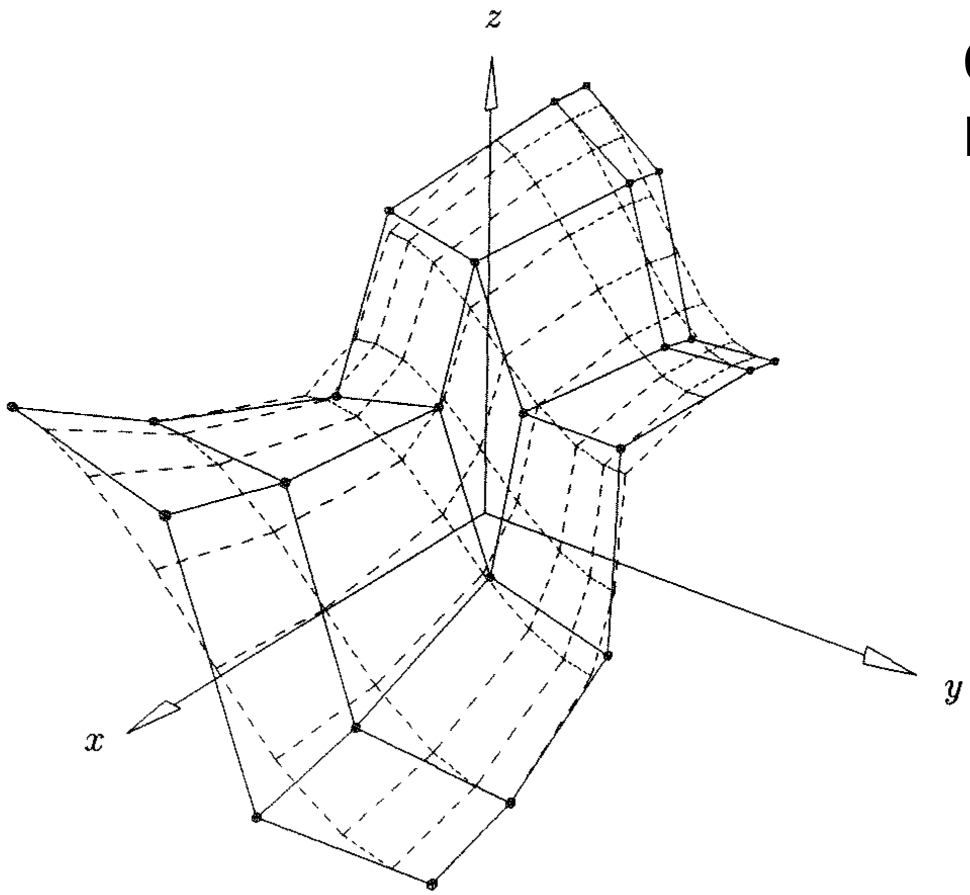
TOL=0.05

surface



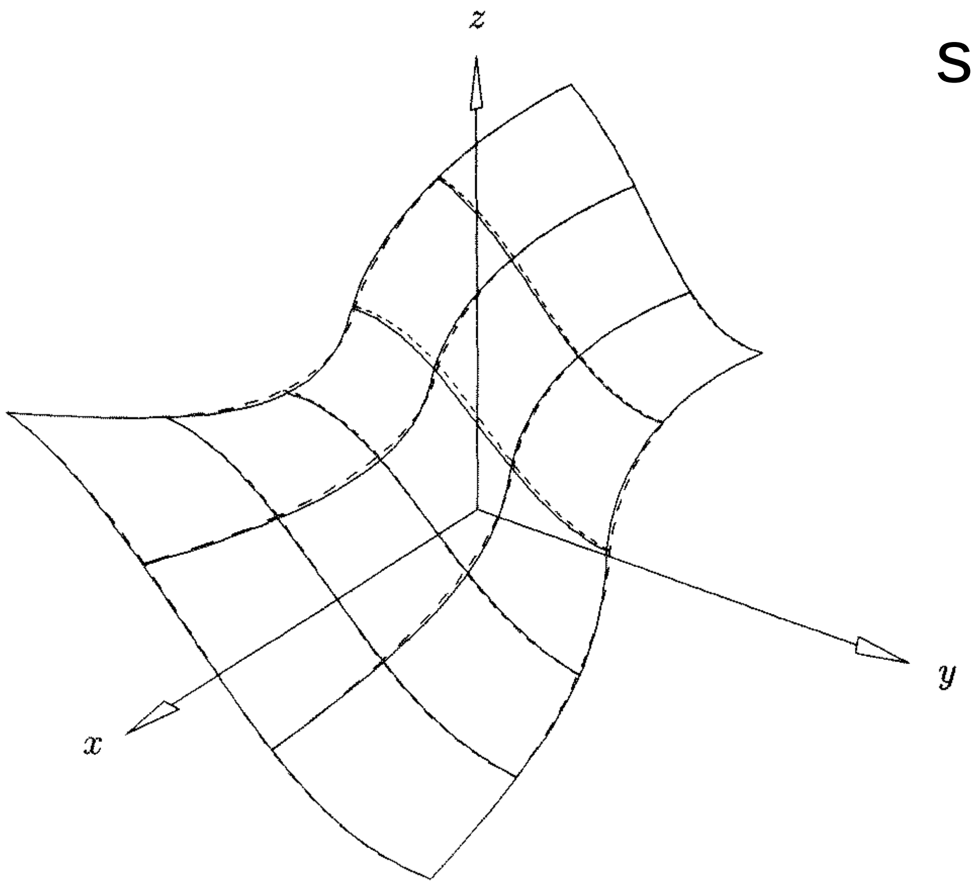
TOL=0.05

control net

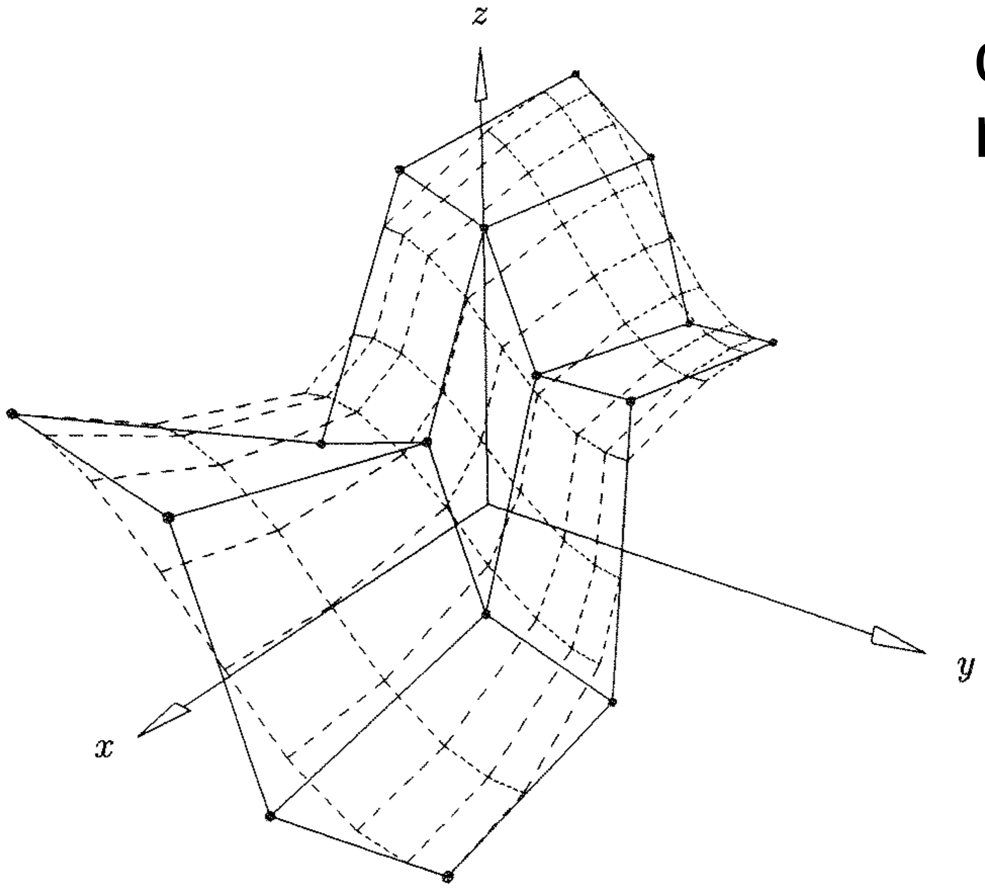


TOL=0.3

surface



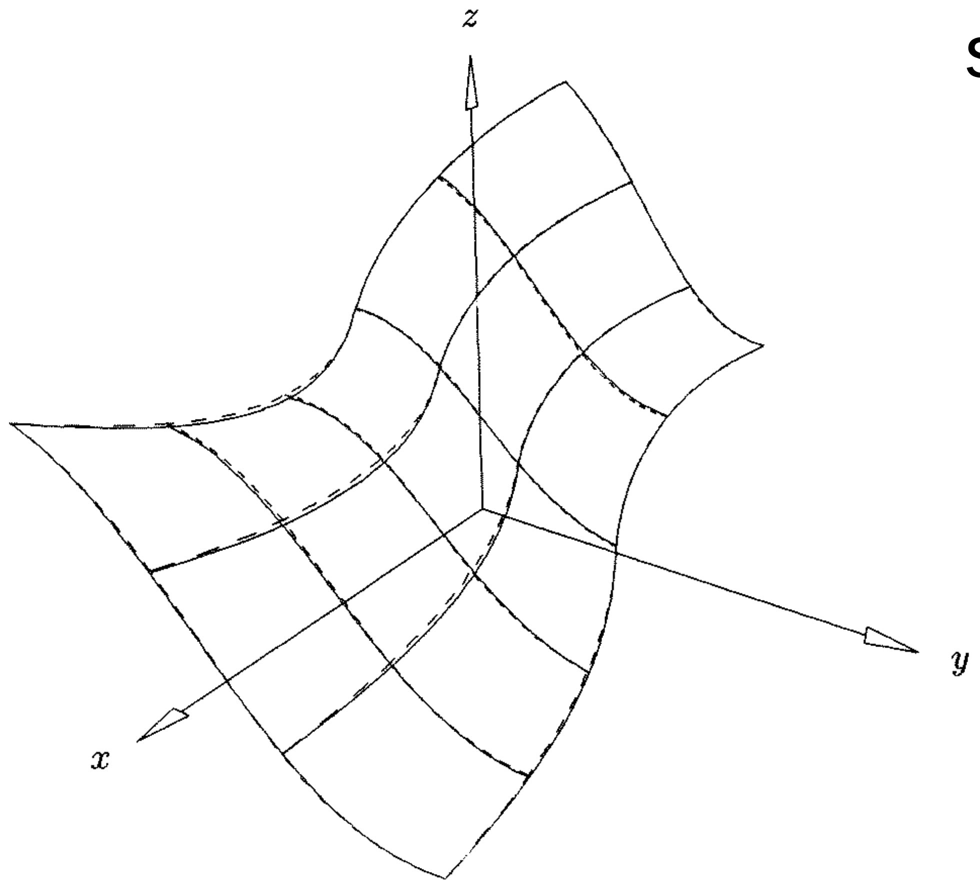
TOL=0.3



control
net

TOL=0.5

surface



TOL=0.5