Appendix A

Pseudo-code of Described Algorithms

A.1 Thinning

The idea of thinning a region is to successively remove pixels from its boundary under the constraint of maintaining its connectedness. Therefore, pixels that connect different segments of the area must not be removed. For a given pixel, we will determine the number of segments it connects. If this is only one segment, then the pixel can be removed. To calculate the number, a simple observation is needed: When going once around the pixel and counting the number of transitions between regions belonging to the area and regions not belonging to the area, the number of transitions is twice the number of connected segments. This relationship is illustrated in figure A.1a). When labeling the neighboring pixels according to figure A.1b) and assuming an 8-connectedness, the pseudo code of procedure 3 performs the thinning. The symbol **&&** expresses a logical AND-operation and the symbol **!** expresses a logical NOT-operation. The sum of boolean values has to be interpreted as the sum of the respective values 0 or 1.

A.2 Smoothing the Line Contours

During feature recognition, the line contours are smoothed before splitting them at high-curvature points. We apply the following method:

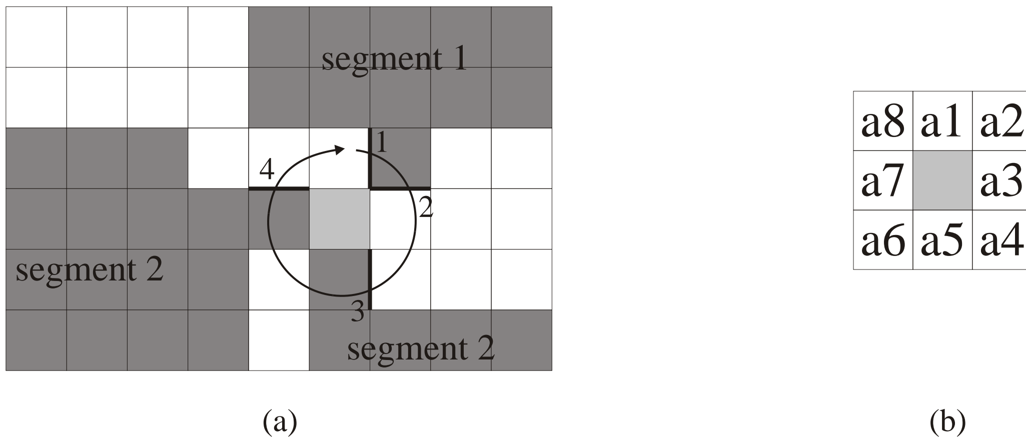

Figure A.1: Determining the crossing number.

Procedure 3 *Thinning()*

Input:	*array I*	*the pixel array, the value 1 represents the area to thin, 0 the background*
Output:	*array I*	*the image with the thinned regions*

```
{
   bool finished;
   do{
     finished=true;
     for (all pixels p in I){
        if (p>0){
           Get the values of the 8 neighboring pixels into a1,a2,...,a8
           byte sigma=a1+a2+a3+a4+a5+a6+a7+a8;
           if (sigma>1){
              byte chi=(a1!=a3)+(a3!=a5)+(a5!=a7)+(a7!=a1)+
                       2*((!a1&&a2&&!a1)+(!a3&&a4&&!a5)+
                          (!a5&&a6&&!a7)+(!a7&&a8&&!a5));
              if(chi==2){
                 p=0; //remove pixel
                 finished=false;
              }
           }
        }
     }
   }while (!finished)
}
```

A.3 Calculation of a Curvature Measure

During feature recognition, a curvature measure is calculated at each point of the line contours.

Procedure 4 *SmoothLine()*

Input:	*array p*	*the array of points of the actual line*
	nP	*the number of points*
Comments:	*V2*	*is the type for a 2D-point and vector.*

```
{
    float q_0 =0.5f;
    float q_m1=0.25f;
    float q_p1=0.25f;
    for (int i=1;i<nP-1;i++){
       p[i]=q_m1*p[i-1]+q_0*p[i]+q_p1*p[i+1];
    }
}
```

Procedure 5 *CalculateLocalCurvatureMeasure()*

Input:	*array p*	*the array of points of the actual line*
	nP	*the number of points*
	wHalf	*the number of points to interleave in both directions from the actual point for the calculation*
	of the angle	
Output:	*array curvature*	*the array to retrieve the curvatures*
Comments:	*V2*	*is the type for a 2D-point and vector.*
	PolarAngleOf(v)	*yields the polar angle of vector v.*
	AngleFromTo(α, β)	*yields the angle ($\in [-\pi, ..., \pi]$) by which α has to be rotated to equal β. The shortest direction*
	for rotation is chosen.	

```
{
    //Pad the first and last entries with zeros, since the curvature
    //of the corresponding points can not be calculated...
    for (int k=0;k<wHalf;k++) curvature[k]=0;
    for (k=nP-wHalf;k<nP;k++) curvature[k]=0;
    //Calculate the curvatures for the middle part...
    for (int i=wHalf;i<nP-wHalf;i++){
         int ia:=i-wHalf;
         int ib:=i+wHalf;
         V2 m:=p[i];
         V2 va:=m-p[ia];
         V2 vb:=p[ib]-m;
         curvature[i]:=AngleFromTo(PolarAngleOf(va),PolarAngleOf(vb));
    }
}
```

A.4 Extracting Local Maxima in Curvature

Once the curvature measure is calculated for all line contours, the goal is to identify points with local maxima in curvature. The following procedure performs this calculation:

Procedure 6 *GetLocalCurvatureMaxima()*

Input:	*nMax*	*the maximum number of extrema the array pos can store*
	curvature	*the array of curvature values as computed by procedure 5*
	nC	*the number of curvature values (equals the number of points of the line)*
	thres	*the curvature threshold*
	minDistance	*the minimal allowed index distance between two intervals*
Output:	*pos*	*the array to retrieve the indices of the local maxima*
	return value	*the number of extrema*
Comments:	*thres*= 0.5, *minDistance*= 3 *in our implementation*	

```
{
   float indexOfActualExtremum=-minDistance-1; //the index of the actual
                                               //extremum
   int nExtrema=0;                             //the number of extrema
   float bestCurvature=0;                      //the actual best curvature
   bool foundAnyExtrema=false;
   for (int i=0;i<nC;i++){
      float aktCurvature=fabs(curvature[i]);
      if (aktCurvature>thres){                 //if the curvature exceeds the
                                               //given threshold
         float dist=i-indexOfActualExtremum;
         if (dist>minDistance){                //open a new extremum
            if (foundAnyExtrema){
               if (nExtrema<nMax) pos[nExtrema++]=indexOfActualExtremum;
            }
            bestCurvature=aktCurvature;
            indexOfActualExtremum=i;
            foundAnyExtrema=true;
         }else{
             if (aktCurvature>bestCurvature){
               bestCurvature=aktCurvature;
               indexOfActualExtremum=i;
            }
         }
      }
   }
   if (foundAnyExtema){
        if (nExtrema<nMax) pos[nExtrema++]=indexOfActualExtremum;
   }
   return nExtrema;
}
```