2-D integration with moving boundaries

Technische
Universiteit
Eindhoven

Instituut Wiskundige Dienstverlening Eindhoven

Report IWDE 92-13

2-D Integration with moving boundaries

J. Spoelstra

December 1992

Den Dolech 2
Postbus 513
5600 MB Eindhoven

INSTITUUT WISKUNDIGE DIENSTVERLENING EINDHOVEN

Report IWDE 92 — 13

2-D Integration with moving boundaries

J. SPOELSTRA

December 1992

1. Introduction

In this report we first discuss the integration technique implemented at present in a program used by the faculty of *Werktuigbouw* of the TUE. In section 3 we consider alternative techniques, with details concerning that method which we advise. We discuss the ideas behind the programs which we have written. In section 4 we mention the optimization technique we would advise for use for first trial runs of the program, which, albeit slow, has the advantage of being very robust and reliable.

Four programs are appended to this report. The first, under the heading *Appendix 1*, is a program to subdivide an area given in polar co-ordinates into small triangles, and to get a visualisation of the area. The second, *Appendix 2*, does the same for an area given in x-y-co-ordinates.

Although it was not part of the agreement for the project, we give some attention to the total structure of the program. The basic elements of a possible main program are discussed in *Appendix 3* in the forms of remarks in a listing of the basic structure of such a program. Part of this is a means of computing the function to be optimized. Parts of a possible program were abstracted and comments made about it, including ideas for computing the line integrals and doing the time-integration. In the last program, *Appendix 4*, we give the actual program for integrating over the two-dimensional area.

2. Analysis of present integration technique

As part of a larger program an integration over a two-dimensional region has to be carried out. The integrand can be computed at a finite number of pre-defined points in the region. At later time intervals, the integrand has to be computed again. In this new integration the previously used points have moved to new positions. The new positions define the shape of the region. Some points may have moved to positions where they do are no longer elements of the regions over which must be integrated.

In the program used at present the integration is carried out by integrating over two-dimensional splines through the concerned points. In this integration either a trapesium-rule or a Simpson-rule is used, and the integration carried out by line, i.e. numerical integration in one direction at a time [Dahlquist & Björk, 1974: 318–9, Press et al, 1989: 144–8]. In the implementation of both these methods, the step interval is shortened until a certain accuracy is reached.

Because both these rules would integrate a low-order polynomial exactly, and the splines are (usually) low order polynomials, no further accuracy will be obtained by the numerical integration methods on decreasing the intervals beyond a certain step length, if the same splines are being used. To overcome this limitation, one possibility is to work with splines on increasingly smaller regions. This, however, can only be done at a large expense of time.

3. An alternative technique for the integration

The most direct alternative approach to a two-dimensional integration of the above type, is subdivision of the concerned region into subregions. The one possibility is to divide the region into rectangles (the easiest method to apply) but unfortunately the rectangles do not remain rectangles as the region changes shape. To work with the original points would therefore need transforming the distorted regions into rectangles as part of the integration procedure. This would need splines through the edges, and subdividing the distorted subregion into two triangles (as the "parallell" edges have different splines passing through them, they can't be considered as a transformed rectangle under a transformation), and the Jacobian would have to be computed from the surfaces of the two triangles.

We have therefore adapted an approach based on subdivision of the region into triangular regions. Triangles remain triangles under continuous transformations, and retain the same relative orientation under such transformations. Using this division, record has to be kept of the positions of the points forming the triangles, and the integrand has to be computed in each of them. Higher accuracy would be obtained in using points halfway between the points on the edges of the triangles, but such points would not remain on the edges under a transformation. Information purely about the points and the value of the integrand in these points have to be used.

In this method the integral over any small triangle is simply the area of the triangle times the average of the integrand at the three vertices of the triangle.

The integral over the whole region is then obtained by summing the integral over all subregions.

The advantage of this approach is that the triangles can approximate the shape of region to a high degree of accuracy, and that the integration can be carried out in a very short time. The disadvantage is that a large number of points have to be used to obtain an accurate approximation to the integral.

Programs and units (in TurboPascal) have been written to implement the above ideas. In one program the subdivision of the region into triangles is done. This information can be stored in a file for all following purposes. This program uses input from a file defining the original region. If some boundaries are not simple straight lines in the co-ordinate system used, functions defining these boundaries can easily be added, as shown for the example in the program. Although not strictly necessary, separate programs have been written for rectangular and polar co-ordinates, so as too ensure that the triangles describe the regions well.

These two programs can be run completely independently of the integration routine, as their only function is to create an initial subdivision of a region in small triangles. The basic idea of the algorithm employed is that the triangle with the longest side must be split into two triangles, until the longest side of all the triangles is below a certain criterium.

A unit was written to compute the integral. For efficient use in the larger program, this contains a procedure for the region specified in terms of rectangular (x, y)-co-ordinates; and one for a region specified in terms of polar co-ordinates (r, ϕ). This was done in order to allow different functions for the integrand to be used in the same program in a simple way.

The programs for subdividing is called by referring to a file in which the original domain is defined, and the functions for integration by referring to the output files from these programs. The forms of these files are fully explained in the listing of the programs.

4. An optimization routine

An optimization technique which is slow, but extremely robust in not being affected be local extrema, or gradients close to zero, is the method of Nelder & Mead [1965: 308]. This method is implemented in the routine AMOEBA in the book and programming package by Press *et al* [1989].

If the program works well with this method, we would consider Powell's method [Press et al, 1989: 331-339] as a means of increasing the speed. Because one of the variables used in the modelling, and over which we have to minimize, also determine the shape of the region (variable a), using an optimization method needing derivatives does not seem advisable, and may add to the computational burden rather than decrease it.

APPENDICES

Appendix 1: Division of an area (given in terms of polar co-ordinates) into triangles.

Appendix 2: Division of an area (given in terms of (x, y)-co-ordinates) into triangles.

Appendix 3: The structure of a possible program.

Appendix 4: The 2-D integration routine.

REFERENCES

Dahlquist G & Björk A. 1974. *Numerical Methods.* Englewood Cliffs: Prentice Hall.

Press WH, Flannery BP, Teukolsky SA & Vetterling WT. 1989. *Numerical Recipes in Pascal.* Cambridge: Cambridge University Press.

Nelder JA & Mead R. 1965. *Computer Journal.* Vol 7, p308.

```
PROGRAM TRIPOLJS;

{  TRIANGULAR COVERING
     of a two-dimensional region, given an original division of the
     region in a (few) triangles.
        IWDE,  Technical University of Eindhoven

     The original division of the region must be supplied by the
     user, given in the following form:

     The data must be in an ASCII-file, in the following format
     (numbers on one line must be separated by blanks)

    Line 1 :
            a)  The number(aantal) of vertices;
            b)  The number(aantal) of triangles;
            c)  The longest triangle-side allowed.
    Line 2
       to
    Line(number of vertices+1) :
            a)  The   r  -co-ordinate of a vertice;
            b)  The theta-co-ordinate of the vertice.

    Line(number of vertices + 2)
       to
    Line(number of vertices + 2 + number of triangles) :
            a)  The number(nummer) of vertice A of a triangle
            b)  The number(nummer) of vertice B of the triangle
            c)  The number(nummer) of vertice C of the triangle.

   EXAMPLE
      4   2  0.5      ::  4 vertices   2 triangles
      0.0  0.0        ::  vertice no 1
      1.0  0.0        ::  vertice no 2
      1.0  1.0        ::  vertice no 3
      0.0  1.0        ::  vertice no 4
       1  2  3        ::  Numbers of vertices A, B and C of triangle 1
       1  4  3        ::  Numbers of vertices A, B and C of triangle 2
}
uses
  crt, Graph;

CONST
    eps    = 1e-6;

TYPE
   real         = single;
   pointpointer = ^points2D;
   Points2D     = object
                    num : word;
                    r,th: real;
                    Ppt : pointpointer;
                          {Points to the next point in the list}
                    FUNCTION xco:real;
                          {Computes the x-co-ordinate of the points}
                    FUNCTION yco:real;
                          {Computes the y-co-ordinate of the points}
                  end;
   abc          = (a,b,c);
   VerticeTags  = record
                     a,b,c : pointpointer;
                  end;
```

```
   SideLengths  = record
                     a,b,c  : real;
                  end;
   Tripointer   = ^drhkdlr;
   drhkdlr      = object {driehoekdeler}
                     Vertice : VerticeTags;
                     side    : sidelengths;
                     tag     : abc;
                     Tpt     : Tripointer;
                     FUNCTION  longest:real;
                             {Computes longest sidelength of triangle}
                     PROCEDURE CompSides;
                             {Compute sidelenghts and store them under
                              "side.a" "side.b" and "side.c"}
                  end;

VAR
  pp : pointer;
  grDriver,grMode,ErrCode                  : integer;
  ax,ay : integer;
   maksx,maksy                          : word;
   ii                                       : word;
   xmaler,ymaler,criterium,longst           : real;
   invoer                                   : text;
   ss                                       : string;
   enough                                   : boolean;
   Firstpoint,Lastpoint,Voorlastpoint       : pointpointer;
              {Pointers to the first and last points in list.
               Firstpoint remains unchanged,
               Lastpoint changes as points are added}
   Firsttri,Lasttri,Voorlasttri,nspl        : Tripointer;
              {Pointer to the first and last triangles in list.}

PROCEDURE addpoint;
  BEGIN
    Voorlastpoint      := lastpoint;
    new(Lastpoint);
    Voorlastpoint^.Ppt := Lastpoint;
    Lastpoint^.num     := Voorlastpoint^.num + 1 ;
    Lastpoint^.Ppt     := nil;
  END;


PROCEDURE addTriangle;
  BEGIN
    Voorlasttri      := lasttri;
    new(lasttri);
    if (memavail<5000) then begin enough := true;
                                  outtextxy(500,400,' TE MIN GEHEUE');
                                  readln;
                                  closegraph;
                                  HALT;
                            end;
    Voorlasttri^.Tpt := Lasttri;
    Lasttri^.Tpt     := nil;
  END;
```

```
PROCEDURE FindPointerToNumbers(VAR Vuma,vumb,vumc : word;
                               VAR VP             : verticetags);
  VAR
    ii : Pointpointer;
  BEGIN
    ii := firstpoint;
    REPEAT
       IF ii^.num = vuma then VP.a := ii;
       IF ii^.num = vumb then VP.b := ii;
       IF ii^.num = vumc then VP.c := ii;
       ii := ii^.Ppt;
    UNTIL ii=nil;
  END;


FUNCTION points2D.xco : real;
  BEGIN
    xco := r*cos(th);
  END;


FUNCTION points2D.yco : real;
  BEGIN
    yco := r*sin(th);
  END;


FUNCTION  drhkdlr.longest : real;
  BEGIN
     CASE tag OF
              a : longest := side.a;
              b : longest := side.b;
              c : longest := side.c;
         END;
  END;


PROCEDURE drhkdlr.CompSides;
  BEGIN
        side.a := sqrt( sqr(vertice.b^.r) + sqr(vertice.c^.r)
                      - 2 * vertice.b^.r * vertice.c^.r
                         * cos(vertice.b^.th - vertice.c^.th) );
        side.b := sqrt( sqr(vertice.a^.r) + sqr(vertice.c^.r)
                      - 2 * vertice.a^.r * vertice.c^.r
                         * cos(vertice.a^.th - vertice.c^.th) );
        side.c :=  sqrt( sqr(vertice.a^.r) + sqr(vertice.b^.r)
                      - 2 * vertice.a^.r * vertice.b^.r
                         * cos(vertice.a^.th - vertice.b^.th) );
        IF (side.a>side.b) AND (side.a>side.c)
           THEN tag := a
           ELSE IF side.b>side.c
                THEN tag := b
                ELSE tag := c;
  END;
```

```
PROCEDURE HalveSideOpp(     TriToSplit : Tripointer;
                        VAR Newpoint   : points2D);
  BEGIN
    WITH TriToSplit^ DO
        CASE tag  OF
          a : Begin
                NewPoint.r := (vertice.b^.r + vertice.c^.r)/2;
                NewPoint.th:= (vertice.b^.th + vertice.c^.th)/2;
              End;
          b : Begin
                NewPoint.r := ( vertice.a^.r+ vertice.c^.r)/2;
                NewPoint.th:= ( vertice.a^.th+ vertice.c^.th)/2;
              End;
          c : Begin
                NewPoint.r := ( vertice.b^.r+ vertice.a^.r)/2;
                NewPoint.th:= ( vertice.b^.th+ vertice.a^.th)/2;
              End;
         END; {case}

    {XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX}

           {The following is an example of a method to use when the
            region to be considered does not have straight sides in the
            r-theta-plane}

           {Specifically for region I}
    IF (Newpoint.r * cos(pi/4-Newpoint.th) > 39)
       THEN
         BEGIN
            Newpoint.r := 39/cos(pi/4-Newpoint.th);
         END;
    {XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX}

  END;



FUNCTION PointerToNewPoint( Newpoint   : points2D;
                            TritoSplit:tripointer   ) : pointpointer;
  VAR
    ipoint,nnew : Pointpointer;
  LABEL 99;
  BEGIN
    ipoint := Firstpoint;
    REPEAT
     IF (abs(ipoint^.r-NewPoint.r)<eps) AND (abs(ipoint^.th-NewPoint.th)<eps)
        THEN Begin
                nnew := ipoint;
                GOTO 99;
             End;
       ipoint := ipoint^.Ppt
    UNTIL ipoint = nil;
    Addpoint;
    nnew := Lastpoint;
    nnew^.r := Newpoint.r;
    nnew^.th := Newpoint.th;
      lynaf( tritosplit, nnew);
  99: pointertonewpoint := nnew;
  END;
```

```
PROCEDURE SplitTriangle(TriToSplit : Tripointer);
  VAR
    Newpoint : points2D;
    nn : Pointpointer;
  BEGIN
    AddTriangle;
    HalveSideOpp(TriToSplit, Newpoint );
    nn := PointertoNewPoint(Newpoint,TriToSplit);
    WITH TriToSplit^ DO
     BEGIN
       LastTri^.vertice := vertice;
       CASE tag OF
         a : BEGIN
              vertice.c := nn;
              LastTri^.vertice.b    := nn;
              lyn(vertice.a^,vertice.c^);
             END;
         b : BEGIN
              vertice.c := nn;
              LastTri^.vertice.a    := nn;
              lyn(vertice.b^,vertice.c^);
             END;
         c : BEGIN
              vertice.a := nn;
              LastTri^.vertice.b    := nn;
              lyn(vertice.a^,vertice.c^);
             END;
        END;{case}
       Compsides;
     END; {with}
    LastTri^.Compsides;
  END;
```

```
FUNCTION SplitWhat(VAR langst : real) : Tripointer;
  VAR
    itri,imax : Tripointer;
    max       : real;
  BEGIN
    imax := FirstTri;
    max  := FirstTri^.longest;
    itri := FirstTri;
    REPEAT
      IF itri^.longest > imax^.longest THEN
         BEGIN
           imax := itri;
           max  := itri^.longest;
         END;
      itri := itri^.Tpt;
    UNTIL itri = nil;
    langst := max;
    SplitWhat := imax;
  END;
```

```
PROCEDURE READPOINTS(var invoer:text; numpoints:word);
   VAR
       smallpointx,bigpointx,smallpointy, bigpointy : Pointpointer;
       itri : Pointpointer;
       ii : word;
       ss1 : string;
   BEGIN
     readln(invoer, Firstpoint^.r,  Firstpoint^.th);
     FOR ii := 2 TO NumPoints DO
       BEgin
        addpoint;
        WITH lastpoint^ DO
          begin
           readln(invoer, r,  th);
          end;
       ENd;
     smallpointx := firstpoint;
     bigpointx   := firstpoint;
     smallpointy := firstpoint;
     bigpointy   := firstpoint;
     itri        := firstpoint;
     REPEAT
       if itri^.xco < smallpointx^.xco then smallpointx := itri;
       if itri^.xco > bigpointx^.xco   then bigpointx   := itri;
       if itri^.yco < smallpointy^.yco then smallpointy := itri;
       if itri^.yco > bigpointy^.yco   then bigpointy   := itri;
       itri := itri^.Ppt;
     UNTIL itri = nil;
     ymaler :=(maksy-20-10)/(bigpointy^.yco-smallpointy^.yco)*0.9;
     xmaler :=(maksx-20-10)/(bigpointx^.xco-smallpointx^.xco)*0.9;
{*}  if  xmaler <= ymaler then ymaler := xmaler
                          else xmaler := ymaler;
     ax     := 10+round(10-xmaler*smallpointx^.xco);
     ay     := 10+round(20+ymaler*bigpointy^.yco);
     criterium := criterium
                  *sqrt( sqr(bigpointx^.xco-smallpointx^.xco)
                       +sqr(bigpointy^.yco-smallpointy^.yco));
     str(criterium:10:3,ss);
     ss := 'criterium ='+ss;
     outtextxy(10,40,ss);
              str(smallpointx^.xco:4:1,ss1);  ss := '('+ss1+';' ;
              str(smallpointy^.yco:4:1,ss1);  ss := ss+ss1+')' ;
              outtextxy(xt(smallpointx^.xco)-20,yt(smallpointy^.yco)+5,ss);
              str(bigpointx^.xco:5:1,ss1);  ss := '('+ss1+';' ;
              str(bigpointy^.yco:5:1,ss1);  ss := ss+ss1+')' ;
              outtextxy(xt(bigpointx^.xco)-50,yt(bigpointy^.yco)-10,ss);
   END;
```

```
PROCEDURE READTRIANGLES(var invoer:text; numtriangles:word);
  VAR
   ii,vnuma,vnumb,vnumc : word;
   VP : verticetags;
  BEGIN
     new(FirstTri);
     FirstTri^.Tpt := nil;
     LastTri := FirstTri;
     WITH lasttri^ DO
       BEGIN
         READLN(invoer, Vnuma, vnumb, vnumc);
         FindPointerToNumbers(Vnuma,vnumb,vnumc, VP);
         vertice := VP;
         CompSides;
             lyn(vertice.a^,vertice.b^);
             lyn(vertice.a^,vertice.c^);
             lyn(vertice.c^,vertice.b^);
       END;
     FOR ii := 2 to NumTriangles DO
       BEgin
         addtriangle;
         WITH lasttri^ DO
           BEGIN
             READLN(invoer, Vnuma, vnumb, vnumc);
             FindPointerToNumbers(Vnuma,vnumb,vnumc, VP);
             vertice := VP;
             CompSides;
             lyn(vertice.a^,vertice.b^);
             lyn(vertice.a^,vertice.c^);
             lyn(vertice.c^,vertice.b^);
           END;
       END;
   END;
```

```
PROCEDURE LEES(naam:string);
VAR
  numpoints,numtriangles : word;
  invoer : text;
BEGIN
     new(Firstpoint);
     Firstpoint^.Ppt := nil;
     Firstpoint^.num := 1;
     Lastpoint := Firstpoint;
     assign(invoer, naam); reset(invoer);
     readln(invoer, NumPoints, NumTriangles, criterium);
     readpoints(invoer,numpoints);
     readtriangles(invoer,numtriangles);
     outtextxy(280,0,'Original figure.  Press ENTER to continue');
     close(invoer);
     readln;
  END;
```

```
PROCEDURE SKRYF(naam:string);
VAR
  ii           : pointpointer;
  NumTriangles : word;
  itri         : tripointer;
  uitvoer      : text;
BEGIN
     assign(uitvoer, naam); rewrite(uitvoer);
     itri := FirstTri;
     numtriangles := 0;
     REPEAT
       inc(numtriangles);
       itri := itri^.Tpt;
     UNTIL itri = nil;
     writeln(uitvoer, Lastpoint^.num:10, NumTriangles:10);
     ii := Firstpoint;
     REPEAT
       with ii^ DO
          writeln(uitvoer, r:12:6,'  ',  th:12:6, '  ', sqr(r)*cos(th):12:6);
       ii := ii^.Ppt;
     UNTIL ii = nil;
     itri := FirstTri;
     REPEAT
       WITH itri^ DO
           writeLN(uitvoer,
             vertice.a^.num:12,' ',vertice.b^.num:12,' ', vertice.c^.num:12);
           itri := itri^.Tpt;
     UNTIL itri = nil;
     close(uitvoer);
  END;


   {A few graphics routines follows}

PROCEDURE grafreg;
BEGIN
  grDriver := Detect;
  InitGraph(grDriver,grMode,'c:\tp6\bgi\');
  ErrCode := GraphResult;
  if ErrCode = grOk then
    begin
      maksx:=GetMaxX;
      maksy:=GetmaxY;
    end
  else
    WriteLn('Graphics error:',GraphErrorMsg(ErrCode));
END;

FUNCTION xt(x:real):word;
  BEGIN
     xt := round(ax +  x*xmaler);
  END;

FUNCTION yt(y:real):word;
  BEGIN
     yt := round(ay -  y*ymaler) ;
  END;

PROCEDURE lyn(pt1,pt2:points2D);
  BEGIN
     line(xt(pt1.xco), yt(pt1.yco),
          xt(pt2.xco), yt(pt2.yco));
  END;
```

```
PROCEDURE lynaf( tritosplit:Tripointer; npt : pointpointer);
  VAR
     bere : word;
  BEGIN
    bere := GetColor;
    Setcolor(GetBkColor);
    WITH tritosplit^ DO
       CASE tag OF
                  a : BEGIN
                         lyn(vertice.b^,vertice.c^);
                         SetColor(Bere);
                         lyn(vertice.b^,npt^);
                         lyn(npt^,vertice.c^);
                      END;
                  b : BEGIN
                         lyn(vertice.a^,vertice.c^);
                         SetColor(Bere);
                         lyn(vertice.a^,npt^);
                         lyn(npt^,vertice.c^);
                      END;
                  c : BEGIN
                         lyn(vertice.b^,vertice.a^);
                         SetColor(Bere);
                         lyn(vertice.b^,npt^);
                         lyn(npt^,vertice.a^);
                      END;
                END;
  END;

procedure kyk(pt : integer);
VAR
   ss1 : string;
   Fillinfo : Fillsettingstype;
BEGIN
     GetFillsettings(Fillinfo);
     setfillstyle(Fillinfo.pattern, GetBkcolor);
     bar(20,20, 250,32);
     str(pt:10,ss);
{     str(pt.th:10:5,ss1);
     ss := ss + ' ' + ss1;}
     outtextxy(20,20,ss);
     setfillstyle(Fillinfo.pattern, fillinfo.color);
END;
```

```
PROCEDURE TRIANGPOLAR(naam1,naam2 :string);
var sj: char;
BEGIN
  grafreg;
  mark(pp);
  lees(naam1);
  enough := false;
  nspl := SplitWhat( longst);
  while not(enough) DO
    BEGIN
       SplitTriangle( nspl );
{       delay(200);}
        nspl := SplitWhat(longst);
{        repeat until keypressed;
        sj := readkey;}
       if longst<criterium then enough := true;
    END;
  skryf(naam2);
  outtextxy(0,0,'PRESS ENTER');
  readln;
  closegraph;
  release(pp);
END;


      {The program itself consist of the following line;
       calling the previous routine with the names of the relevant
       files}

BEGIN
  triangpolar('a:tri_geb1.inv', 'a:tri_geb1.uit' );
end.
```

```
    {The structure of this program is almos identical to the
     previous one, and comments are therefore only given where
     differences occur.}


PROGRAM trianjs;

{  TRIANGULAR COVERING
     of a two-dimensional region, given an original division of the
     region in a (few) triangles.
        IWDE,  Technical University of Eindhoven

     The original division of the region must be supplied by the
     user, given in the following form:

     The data must be in an ASCII-file, in the following format
     (numbers on one line must be separated by blanks)

    Line 1 :
            a)  The number(aantal) of vertices;
            b)  The number(aantal) of triangles;
            c)  The longest triangle-side allowed.
    Line 2
       to
    Line(number of vertices+1) :
            a)  The x-co-ordinate of a vertice (or the r-coordinate);
            b)  The y-co-ordinate of the vertice (or the -coordinate).

    Line(number of vertices + 2)
       to
    Line(number of vertices + 2 + number of triangles) :
            a)  The number(nummer) of vertice A of a triangle
            b)  The number(nummer) of vertice B of the triangle
            c)  The number(nummer) of vertice C of the triangle.

   EXAMPLE
      4   2  0.5      ::  4 vertices   2 triangles
      0.0  0.0        ::  vertice no 1
      1.0  0.0        ::  vertice no 2
      1.0  1.0        ::  vertice no 3
      0.0  1.0        ::  vertice no 4
       1  2  3        ::  tag of vertices A, B and C of triangle 1
       1  4  3        ::  tag of vertices A, B and C of triangle 2
}
uses
  crt, Graph;
CONST
    eps    = 1e-6;
    skaal = 0.9;
TYPE
   real         = single;
   pointpointer = ^points2D;
   Points2D     = record
                     num : word;
                     x,y : real;
                     Ppt : pointpointer;
                  end;
   abc          = (a,b,c);
   VerticeTags  = record
                     a,b,c : pointpointer;
                  end;
   SideLengths  = record
                     a,b,c  : real;
                  end;
```

```
   Tripointer   = ^drhkdlr;
   drhkdlr      = object {driehoekdeler}
                    Vertice : VerticeTags;
                    side    : sidelengths;
                    tag     : abc;
                    Tpt     : Tripointer;
                    FUNCTION  longest:real;
                    PROCEDURE CompSides;
                  end;
var
  pp : pointer;
  grDriver,grMode,ErrCode                  : integer;
  ax,ay                                    : integer;
   maksx,maksy                             : word;
   ii                                      : word;
   xmaler,ymaler,criterium,longst          : real;
   invoer                                  : text;
   ss                                      : string;
   enough                                  : boolean;
   Firstpoint,Lastpoint,Voorlastpoint      : pointpointer;
   Firsttri,Lasttri,Voorlasttri,nspl       : Tripointer;

PROCEDURE addpoint;
  BEGIN
    Voorlastpoint      := lastpoint;
    new(Lastpoint);
    Voorlastpoint^.Ppt := Lastpoint;
    Lastpoint^.num     := Voorlastpoint^.num+1;
    Lastpoint^.Ppt     := nil;
  END;
PROCEDURE addTriangle;
  BEGIN
    Voorlasttri      := lasttri;
    new(lasttri);
    if (memavail<10000) then begin enough := true;
                                   outtextxy(200,200,' TE MIN GEHEUE');
                                   HALT;
                             end;
    Voorlasttri^.Tpt := Lasttri;
    Lasttri^.Tpt     := nil;
  END;
PROCEDURE FindPointerToNumbers(VAR Vuma,vumb,vumc : word;
                               VAR VP             : verticetags);
  VAR
    ii : Pointpointer;
  BEGIN
    ii := firstpoint;
    REPEAT
       IF ii^.num = vuma then VP.a := ii;
       IF ii^.num = vumb then VP.b := ii;
       IF ii^.num = vumc then VP.c := ii;
       ii := ii^.Ppt;
    UNTIL ii=nil;
  END;

FUNCTION  drhkdlr.longest : real;
  BEGIN
     CASE tag OF
              a : longest := side.a;
              b : longest := side.b;
              c : longest := side.c;
         END;
  END;
```

```
PROCEDURE drhkdlr.CompSides;
  BEGIN
        side.a := sqrt( sqr(vertice.b^.x - vertice.c^.x)
                      + sqr(vertice.b^.y - vertice.c^.y) );
        side.b := sqrt( sqr(vertice.a^.x - vertice.c^.x)
                      + sqr(vertice.a^.y - vertice.c^.y) );
        side.c := sqrt( sqr(vertice.a^.x - vertice.b^.x)
                      + sqr(vertice.a^.y - vertice.b^.y) );
        IF (side.a>side.b) AND (side.a>side.c)
           THEN tag := a
           ELSE IF side.b>side.c
                THEN tag := b
                ELSE tag := c;
  END;


PROCEDURE HalveSideOpp(    TriToSplit : Tripointer;
                       VAR Newpoint   : points2D);
  BEGIN
    WITH TriToSplit^ DO
        CASE tag  OF
          a : Begin
                NewPoint.x := (vertice.b^.x + vertice.c^.x)/2;
                NewPoint.y := (vertice.b^.y + vertice.c^.y)/2;
              End;
          b : Begin
                NewPoint.x := ( vertice.a^.x+ vertice.c^.x)/2;
                NewPoint.y := ( vertice.a^.y+ vertice.c^.y)/2;
              End;
          c : Begin
                NewPoint.x := ( vertice.b^.x+ vertice.a^.x)/2;
                NewPoint.y := ( vertice.b^.y+ vertice.a^.y)/2;
              End;
         END; {case}
  END;


FUNCTION PointerToNewPoint( Newpoint : points2D  ) : pointpointer;
  VAR
    ipoint,nnew : Pointpointer;
  LABEL 99;
  BEGIN
    ipoint := Firstpoint;
    REPEAT
      IF (abs(ipoint^.x-NewPoint.x)<eps) AND (abs(ipoint^.y-NewPoint.y)<eps)
       THEN Begin
              nnew := ipoint;
              GOTO 99;
            End;
      ipoint := ipoint^.Ppt
    UNTIL ipoint = nil;
    Addpoint;
    nnew := Lastpoint;
    nnew^.x := Newpoint.x;
    nnew^.y := Newpoint.y;
  99: pointertonewpoint := nnew;
  END;
```

```
PROCEDURE SplitTriangle(TriToSplit : Tripointer);
  VAR
    Newpoint : points2D;
    nn : Pointpointer;
  BEGIN
    AddTriangle;
    HalveSideOpp(TriToSplit, Newpoint );
    nn := PointertoNewPoint(Newpoint);
    WITH TriToSplit^ DO
     BEGIN
       LastTri^.vertice := vertice;
       CASE tag OF
         a : BEGIN
              vertice.c := nn;
              LastTri^.vertice.b    := nn;
              line(xt(vertice.a^.x), yt(vertice.a^.y),
                   xt(vertice.c^.x), yt(vertice.c^.y));
             END;
         b : BEGIN
              vertice.c := nn;
              LastTri^.vertice.a    := nn;
              line(xt(vertice.b^.x), yt(vertice.b^.y),
                   xt(vertice.c^.x), yt(vertice.c^.y));
             END;
         c : BEGIN
              vertice.a := nn;
              LastTri^.vertice.b    := nn;
              line(xt(vertice.a^.x),yt(vertice.a^.y),
                   xt(vertice.c^.x),yt(vertice.c^.y));
             END;
        END;{case}
       Compsides;
     END; {with}
    LastTri^.Compsides;
  END;



FUNCTION SplitWhat(VAR langst : real) : Tripointer;
  VAR
    itri,imax : Tripointer;
    max       : real;
  BEGIN
    imax := FirstTri;
    max  := FirstTri^.longest;
    itri := FirstTri;
    REPEAT
      IF itri^.longest > imax^.longest THEN
         BEGIN
           imax := itri;
           max  := itri^.longest;
         END;
      itri := itri^.Tpt;
    UNTIL itri = nil;
    langst := max;
    SplitWhat := imax;
  END;
```

```
PROCEDURE READPOINTS(var invoer:text; numpoints:word);
   VAR
       smallpointx,bigpointx,smallpointy, bigpointy : Pointpointer;
       itri : Pointpointer;
       ii : word;
       ss1 : string;
   BEGIN
     readln(invoer, Firstpoint^.x,  Firstpoint^.y);
     FOR ii := 2 TO NumPoints DO
       BEgin
        addpoint;
        WITH lastpoint^ DO
          begin
           readln(invoer, x,  y);
          end;
       ENd;
     smallpointx := firstpoint;
     bigpointx   := firstpoint;
     smallpointy := firstpoint;
     bigpointy   := firstpoint;
     itri        := firstpoint;
     REPEAT
       if itri^.x < smallpointx^.x then smallpointx := itri;
       if itri^.x > bigpointx^.x   then bigpointx   := itri;
       if itri^.y < smallpointy^.y then smallpointy := itri;
       if itri^.y > bigpointy^.y   then bigpointy   := itri;
       itri := itri^.Ppt;
     UNTIL itri = nil;
     xmaler :=(maksx-20-10)/(bigpointx^.x-smallpointx^.x)*0.9;
     ymaler :=(maksy-20-10)/(bigpointy^.y-smallpointy^.y)*0.9;
{*}  if  xmaler <= ymaler then ymaler := xmaler
                       else xmaler := ymaler;
     ax     := 10+round(10-xmaler*smallpointx^.x);
     ay     := 10+round(20+ymaler*bigpointy^.y);
     criterium := criterium
                  *sqrt( sqr(bigpointx^.x-smallpointx^.x)
                        +sqr(bigpointy^.y-smallpointy^.y));
     str(criterium:10:3,ss);
     ss := 'criterium ='+ss;
     outtextxy(150,10,ss);
              str(smallpointx^.x:4:1,ss1);  ss := '('+ss1+';' ;
              str(smallpointy^.y:4:1,ss1);  ss := ss+ss1+')' ;
              outtextxy(xt(smallpointx^.x)-20,yt(smallpointy^.y)+5,ss);
              str(bigpointx^.x:5:1,ss1);  ss := '('+ss1+';' ;
              str(bigpointy^.y:5:1,ss1);  ss := ss+ss1+')' ;
              outtextxy(xt(bigpointx^.x)-50,yt(bigpointy^.y)-10,ss);
   END;
```

```
PROCEDURE READTRIANGLES(var invoer:text; numtriangles:word);
  VAR
   ii,vnuma,vnumb,vnumc : word;
   VP : verticetags;
  BEGIN
     new(FirstTri);
     FirstTri^.Tpt := nil;
     LastTri := FirstTri;
     WITH lasttri^ DO
       BEGIN
         READLN(invoer, Vnuma, vnumb, vnumc);
         FindPointerToNumbers(Vnuma,vnumb,vnumc, VP);
         vertice := VP;
         CompSides;
             line(xt( vertice.a^.x), yt( vertice.a^.y),
                  xt( vertice.b^.x), yt( vertice.b^.y));
             line(xt( vertice.b^.x), yt( vertice.b^.y),
                  xt( vertice.c^.x), yt( vertice.c^.y));
             line(xt( vertice.c^.x), yt( vertice.c^.y),
                  xt( vertice.a^.x), yt( vertice.a^.y));
       END;
     FOR ii := 2 to NumTriangles DO
       BEgin
         addtriangle;
         WITH lasttri^ DO
           BEGIN
             READLN(invoer, Vnuma, vnumb, vnumc);
             FindPointerToNumbers(Vnuma,vnumb,vnumc, VP);
             vertice := VP;
             CompSides;
             line(xt( vertice.a^.x), yt( vertice.a^.y),xt( vertice.b^.x), yt( vertic
             line(xt( vertice.b^.x), yt( vertice.b^.y),xt( vertice.c^.x), yt( vertic
             line(xt( vertice.c^.x), yt( vertice.c^.y),xt( vertice.a^.x), yt( vertic
           END;
       END;
   END;


PROCEDURE LEES(naam:string);
VAR
  numpoints,numtriangles : word;
  invoer : text;
BEGIN
     new(Firstpoint);
     Firstpoint^.Ppt := nil;
     Firstpoint^.num := 1;
     Lastpoint := Firstpoint;
     assign(invoer, naam); reset(invoer);
     readln(invoer, NumPoints, NumTriangles, criterium);
     readpoints(invoer,numpoints);
     readtriangles(invoer,numtriangles);
     outtextxy(280,0,'Original figure.  Press ENTER to continue');
     close(invoer);
     readln;
  END;
```

```
PROCEDURE SKRYF(naam:string);
VAR
  ii           : pointpointer;
  NumTriangles : word;
  itri         : tripointer;
  uitvoer      : text;
BEGIN
     assign(uitvoer, naam); rewrite(uitvoer);
     itri := FirstTri;
     numtriangles := 0;
     REPEAT
       inc(numtriangles);
       itri := itri^.Tpt;
     UNTIL itri = nil;
     writeln(uitvoer, Lastpoint^.num:10, NumTriangles:10);
     ii := Firstpoint;
     REPEAT
       with ii^ DO
          writeln(uitvoer, x:12:6,'  ',  y:12:6, '  ',sqr(x)+sqr(y):12:6);
       ii := ii^.Ppt;
     UNTIL ii = nil;
     itri := FirstTri;
     REPEAT
       WITH itri^ DO
           writeLN(uitvoer,
             vertice.a^.num:12,' ',vertice.b^.num:12,' ', vertice.c^.num:12);
           itri := itri^.Tpt;
     UNTIL itri = nil;
     close(uitvoer);
  END;

PROCEDURE grafreg;
BEGIN
  grDriver := Detect;
  InitGraph(grDriver,grMode,'c:\tp6\bgi\');
  ErrCode := GraphResult;
  if ErrCode = grOk then
    begin
      maksx:=GetMaxX;
      maksy:=GetmaxY;
    end
  else
    WriteLn('Graphics error:',GraphErrorMsg(ErrCode));
END;

FUNCTION xt(x:real):word;
  BEGIN
     xt := round(ax +  x*xmaler);
  END;
FUNCTION yt(y:real):word;
  BEGIN
     yt := round(ay -  y*ymaler) ;
  END;
```

```
PROCEDURE TRIANGULATOR(naam1,naam2 :string);
var a: char;
BEGIN
  grafreg;
  mark(pp);
  lees(naam1);
  enough := false;
  nspl := SplitWhat( longst);
  while not(enough) DO
    BEGIN
       SplitTriangle( nspl );
{       delay(200);}
       nspl := SplitWhat(longst);
       if longst<criterium then enough := true;
    END;
  skryf(naam2);
  outtextxy(0,0,'PRESS ENTER');
  readln;
  closegraph;
{  writeln(memavail);}
  release(pp);
END;

BEGIN
  triangulator('a:tri_geb2.inv', 'a:tri_geb2.uit' );
END.
```

```
{{{{{                The program HOOF                    }}}}}


PROGRAM HOOF;
USES
   intgenjs, funct, solvede;

        {These units compute the function to be optimized;      }
        {read and write data concerning the mesh, and solve     }
        {the differential equations numerically for all points }

CONST
   mp = 5;
TYPE
   glmpnp = ARRAY [1..mp,1..np] OF real;
   glmp   = ARRAY [1..mp]       OF real;
        {mp and the above array's are needed by the             }
        {optimization-program                                   }

VAR
   ftol        : real;
   iter,ndim   : integer;
   p           : glmpnp;
   yy          : glmp;
   tijd, delt  : real;

PROCEDURE amoeba(VAR  p    : glmpnp;
                 VAR  y    : glmp;
                      ndim : integer;
                      ftol : real;
                 VAR  iter : integer);
        {The procedure AMOEBA needs a function                  }
        {           FUNC(xx:glnp)                               }
        {to be defined. this is done in the UNIT FUNCT          }

       {We now sketch the main program.                         }

       {From a value of the time, the quantities needed by the  }
       {above procedure AMOEBA is initialized. This procedure   }
       {then minimizes FUNC, and returns the values for         }
       {       ALPHA, LAMBDA, BETA and a.                       }
       {With these values known, one time-integration for all   }
       {points on the grid are made by the routines             }
       {       SOLVE_DE_1      and                              }
       {       SOLVE_DE_2,                                      }
       {until the desired time is reached                       }

BEGIN
   tijd := 0;
   REPEAT
     Initialize_variables_for_Amoeba
     amoeba(p, yy, ndim, ftol, iter);
     Solve_DE_1('a:tri_geb1.uit',tijd,delt);
     Solve_DE_2('a:tri_geb2.uit',tijd,delt);
     tijd := tijd + delt;
   UNTIL delt = 10;
END.
        {Although we have written 'a:*.*' for the files in the  }
        {above, using disk   'a:'   is    definitely not        }
        {advised, for reasons of speed. A RAM-drive should be   }
        {created and used for all the files in the program.     }
```

```
{{{{{                The unit FUNC                        }}}}}

        {In this unit the function that has to be optimized    }
        {is defined                                            }


UNIT FUNCT;

INTERFACE
  CONST
     np = 4;
  TYPE
     glnp = ARRAY [1..np] OF real;


  FUNCTION func(pr:glnp):real;


IMPLEMENTATION
  USES
     intgenjs;
  VAR
     alpha, beta, lambda, aaa : real;

        {The following function must be specified by the user. }
        {The three options under the CASE-statement allows the }
        {three different integrals to be evaluated with the    }
        {same trapesium-rule-program with little coding.       }

  FUNCTION f_123(num:shortint; x:real):real;
   BEGIN

        {Functions for the different sections to be inserted   }
        {here in place of the number "1"                       }

      CASE num OF
         1  :  f_123 := 1;
         2  :  f_123 := 1;
         3  :  f_123 := 1;
        END;
  END;

        {In addition to the surface integrals, line integrals  }
        {along the boundaries of some regions have to be       }
        {determined. The following is a routine for this,      }
        {using the trapesium rule and a predetermined number   }
        {of divisions of the interval                          }

PROCEDURE trapzd(numb : shortint;
                 a,b  : real;
                 n    : integer;
                 VAR s: real);

        {For full particulars of the program TRAPZD, see the   }
        {full listing of the programs following                }

        {In the first two of the following three integrations, }
        {the only complication is that the upper bounds of the }
        {integrals have to be read from the data files used by }
        {the surface-integrals, containing the information     }
        {about the points                                      }
```

```
FUNCTION int_gam_1(naam : string):real;
  VAR
    invoer     : text;
    j          : integer;
    dummy, c_1 : real;
  BEGIN
    assign(invoer,naam); reset(invoer);
    readln(invoer); readln(invoer); readln(invoer);
    readln(invoer,c_1,dummy,dummy);     close(invoer);
    for j := 1 to 6 do
       trapzd(1, 0.0, c_1, j, dummy);
    int_gam_1 := dummy;
  END;


FUNCTION int_gam_2(naam : string):real;
  VAR
    invoer : text;
    j      : integer;
    dummy, c_2 : real;
  BEGIN
    assign(invoer,naam); reset(invoer);
    readln(invoer); readln(invoer); readln(invoer);
    readln(invoer,dummy, c_2, dummy);    close(invoer);
    for j := 1 to 6 do
         trapzd(2, 0.0, c_2, j, dummy);
    int_gam_2 := dummy;
  END;


FUNCTION int_gam_3(aatemp:real):real;
  VAR
    j      : integer;
    dummy  : real;
  BEGIN
    for j := 1 to 6 do
       trapzd(3, 0.0, aatemp, j, dummy);
    int_gam_3 := dummy;
  END;


FUNCTION func{(pr:glnp):real;};
  VAR
    tyd1,tyd2,tyd3,tyd4,tyd5 : real;
  BEGIN
    alpha := pr[1];
    beta  := pr[2];
    lambda:= pr[3];
    aaa   := pr[4];
    tyd1 := integ_geb_1('a:tri_geb1.uit',alpha,lambda,beta);
    tyd2 := integ_geb_2('a:tri_geb2.uit',beta,aaa);
    tyd3 := int_gam_1('a:tri_geb1.uit');
    tyd4 := int_gam_2('a:tri_geb2.uit');
    tyd5 := int_gam_3(aaa);
    func := tyd1 + tyd2 + tyd3 + tyd4 + tyd5;
  END;
```

```
{{{{{                The unit INTGENJS                        }}}}}

UNIT INTGENJS;

        { Double-integration based on a division (supplied by  }
        { the user) of the concerned region into triangles.    }

        { The data must be in an ASCII-file, and the name of   }
        { the file must be passed to the integration-function, }
        { in the form:                                         }
        {                                                      }
        {          INTEGjs('c:\YourDir\Yourname.Ext');         }
        {                                                      }
        { To be able to achieve a relative error of, eg.       }
        { approximately 0.05%, the longest side of any triangle}
        { should be less than 1/40 of the longest dimension of }
        { the original figure, and the criterium should then be}
        { 1/40 = 0.025.                                        }
        { We advice accuracy of 0.1%, which would need a       }
        { criterium of 0.05, and which would need 1/4 the time }
        { and memory of the above.                             }

        { For reasons of efficiency, the data can be stored in }
        { a file on a RAM-drive.                               }

        { The data in the data-file must be in the following   }
        { format (numbers on one line must be separated by     }
        { blanks) (As the meaning of the term "number" can be  }
        { unclear, it has been qualified by the dutch words    }
        { "aantal" or "nummer", depending on whether it refers }
        { to a quantity or an identification number.)          }

    { Line 1  : a)  The number(aantal) of vertices             }
    {           b)  The number(aantal) of triangles            }
    { Line 2                                                   }
    {   to                                                     }
    { Line(number of vertices+1) :                             }
    {        a)  The x-co-ordinate of a point                  }
    {        b)  The y-co-ordinate of the point                }
    {        c)  The value of the function at the point        }
    { Line(number of vertices + 2)                             }
    {  to                                                      }
    { Line(number of vertices + 2 + number of triangles):      }
    {         a)  The number(nummer) of vertice A of a         }
    {             triangle                                     }
    {         b)  The number(nummer) of vertice B of the       }
    {             triangle                                     }
    {         c)  The number(nummer) of vertice C of the       }
    {             triangle                                     }
    {                                                          }
    { EXAMPLE of a data file:                                  }
    {  4   2          ::  4 vertices   2 triangles             }
    {  0.0  0.0  1.0  ::  Vertice no 1, functionvalue there    }
    {  1.0  0.0  1.0  ::  Vertice no 2, functionvalue at       }
    {  1.0  1.0  1.0  ::  Vertice no 3, functionvalue at       }
    {  0.0  1.0  1.0  ::  Vertice no 4, functionvalue at       }
    {   1  2  3       ::  Numbers: vertices A,B,C of triangle 1}
    {   1  4  3       ::  Numbers: vertices A,B,C of triangle 2}
```

```
INTERFACE

        {of intgenjs.unt}

  TYPE
    real         = single;
    pointpointer = ^points2D;
    funcpointer  = ^real;
    Points2D     = record
                     num : word;
                     x,y : real;
                     Ppt : pointpointer;
                     Fpt : funcpointer;
                   end;

        { the TYPE  "POINTS2D" can be thought of as an card in  }
        { a filing cabinet, containing information about a      }
        { point:                                                }
        {    num: A number of the point.                        }
        {    x,y: The coordinates of the point. This can be     }
        {         extended to z, or whatever else may be needed.}
        {    Ppt: A pointer containing the address of the next  }
        {         "card in the filing system". If this contains }
        {         the TurboPascal constant "NIL", it is the last}
        {         "card".                                       }
        {    Fpt: A pointer containing the address of the value }
        {         of the function in this point. This can just  }
        {         as well be declared to be the function value  }
        {         itself instead of a pointer.                  }

    VerticeTags  = record
                     a,b,c : pointpointer;
                   end;
    Tripointer   = ^triangle;
    Triangle     = object
                     Vertice : VerticeTags;
                     Tpt     : Tripointer;
                     FUNCTION Oppervlakte : real;
                     FUNCTION Integraal : real;
                   end;

        { The TYPE  "TRIANGLE" can be thought of as a card in a }
        { filing system containing:                             }
        {    Vertice : pointers to the POINTS2D-records         }
        {              of the three vertices;                   }
        {    Tpt     : the address of the next triangle; and    }
        {    OPPERVLAKTE : the area of the triangle.            }
        {    INTEGRAAL   : The integral over the triangle.      }
        { The last two are only evaluated when needed.          }

  FUNCTION Integ_Geb_1(naam:string; alpha, lambda ,beta:real ):real;

  FUNCTION Integ_Geb_2(naam:string; beta, aaa :real ):real;

  PROCEDURE LEES_PTS_TRIS(naam:string);

  PROCEDURE SKRYF_PTS_TRIS(naam:string);
```

```
IMPLEMENTATION

        {

  VAR
     p : pointer;
     Firstpoint,lastpoint,voorlastpoint : pointpointer;
     Firsttri,  lasttri,  voorlasttri   : tripointer;


  FUNCTION triangle.Oppervlakte : real;
    BEGIN
       Oppervlakte := abs(
         (Vertice.b^.x - Vertice.a^.x)*(Vertice.c^.y - Vertice.a^.y)
      -  (Vertice.b^.y - Vertice.a^.y)*(Vertice.c^.x - Vertice.a^.x) )/2;
    END;


  FUNCTION triangle.Integraal : real;
    BEGIN
        Integraal := oppervlakte *
          (Vertice.a^.Fpt^ + Vertice.b^.Fpt^ + Vertice.c^.Fpt^)/3
    END;


  FUNCTION Double_Integral :  real;
    VAR
       som : real;
       itri   : tripointer;
    BEGIN
        som := 0;
        itri := FirstTri;
        REPEAT
           som := som + itri^.integraal;
           itri := itri^.Tpt;
        UNTIL itri = nil;
        Double_Integral := som;
    END;


  PROCEDURE addpoint;
    BEGIN
      Voorlastpoint      := lastpoint;
      new(Lastpoint);
      Voorlastpoint^.Ppt := Lastpoint;
      Lastpoint^.num     := Voorlastpoint^.num+1;
      Lastpoint^.Ppt     := nil;
    END;


  PROCEDURE addTriangle;
    BEGIN
      Voorlasttri      := lasttri;
      new(lasttri);
      if (memavail<10000) then begin
                               writeln('O gaats, te min plek!');readln;
                               HALT;
                              end;
      Voorlasttri^.Tpt := Lasttri;
      Lasttri^.Tpt     := nil;
    END;
```

```
PROCEDURE FindPointerToNumbers(VAR Vuma,vumb,vumc : word;
                               VAR VP              : verticetags);
  VAR
    ii : Pointpointer;
  BEGIN
    ii := firstpoint;
    REPEAT
       IF ii^.num = vuma then VP.a := ii;
       IF ii^.num = vumb then VP.b := ii;
       IF ii^.num = vumc then VP.c := ii;
       ii := ii^.Ppt;
    UNTIL ii=nil;
  END;


PROCEDURE READPOINTS(var invoer:text; numpoints:word);
   VAR
       ii : word;
   BEGIN
     new(Firstpoint);
     Firstpoint^.Ppt := nil;
     Firstpoint^.num := 1;
     Lastpoint := Firstpoint;
     new(Firstpoint^.Fpt);
     readln(invoer, Firstpoint^.x,  Firstpoint^.y, Firstpoint^.Fpt^);
     FOR ii := 2 TO NumPoints DO
       BEgin
        addpoint;
        WITH lastpoint^ DO
          begin
            new(Fpt);
            readln(invoer, x,  y, Fpt^);
          end;
       ENd;
   END;


PROCEDURE READTRIANGLES(var invoer:text; numtriangles:word);
  VAR
   ii,vnuma,vnumb,vnumc : word;
   VP : verticetags;
  BEGIN
     new(FirstTri);
     FirstTri^.Tpt := nil;
     LastTri := FirstTri;
     WITH lasttri^ DO
       BEGIN
         READLN(invoer, Vnuma, vnumb, vnumc);
         FindPointerToNumbers(Vnuma,vnumb,vnumc, VP);
         vertice := VP;
       END;
     FOR ii := 2 to NumTriangles DO
       BEgin
         addtriangle;
         WITH lasttri^ DO
           BEGIN
             READLN(invoer, Vnuma, vnumb, vnumc);
             FindPointerToNumbers(Vnuma,vnumb,vnumc, VP);
             vertice := VP;
           END;
       END;
  END;
```

```
PROCEDURE LEES_PTS_TRIS(naam:string);
  VAR
    numpoints,numtriangles : word;
    invoer : text;
  BEGIN
     assign(invoer,naam); reset(invoer);
     readln(invoer, NumPoints, NumTriangles);
     readpoints(invoer,numpoints);
     readtriangles(invoer,numtriangles);
     close(invoer);
  END;

PROCEDURE SKRYF_PTS_TRIS(naam:string);
VAR
  ii           : pointpointer;
  NumTriangles : word;
  itri         : tripointer;
  uitvoer      : text;
BEGIN
     assign(uitvoer, naam); rewrite(uitvoer);
     itri := FirstTri;
     numtriangles := 0;
     REPEAT
       inc(numtriangles);
       itri := itri^.Tpt;
     UNTIL itri = nil;
     writeln(uitvoer, Lastpoint^.num:10, NumTriangles:10);
     ii := Firstpoint;
     REPEAT
       with ii^ DO
          writeln(uitvoer, x:12:6,'  ',  y:12:6, '  ',Fpt^:12:6);
       ii := ii^.Ppt;
     UNTIL ii = nil;
     itri := FirstTri;
     REPEAT
       WITH itri^ DO
           writeLN(uitvoer,
             vertice.a^.num:12,' ',vertice.b^.num:12,' ', vertice.c^.num:12);
           itri := itri^.Tpt;
     UNTIL itri = nil;
     close(uitvoer);
END;


FUNCTION f_geb_1(alpha, lambda,beta:real; xx,yy : real):real;  {r en  hier}
  {XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX }
  { Insert integrand for gebied I }
  BEGIN
    f_geb_1 := 1;
  END;
```

```
FUNCTION Integ_Geb_1(naam:string; alpha, lambda, beta :real ):real;
  VAR
    ii : Pointpointer;
  BEGIN
     mark(p);
     LEES_PTS_TRIS(naam);
     ii := firstpoint;
     REPEAT
       ii^.Fpt^ := f_geb_1(alpha, lambda,beta, ii^.x, ii^.y);      {r en  hier}
       ii := ii^.Ppt;
     UNTIL ii=nil;
     Integ_Geb_1 := Double_Integral;
     release(p);
  END;


FUNCTION f_geb_2(beta, aaa :real; xx,yy : real):real;    {r en  hier}
  {XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX }
  { Insert integrand for gebied II+III}
  BEGIN
    IF xx<aaa
     THEN
        f_geb_2 := 1
     ELSE
        f_geb_2 := 0
  END;


FUNCTION Integ_Geb_2(naam:string; beta, aaa :real ):real;
  VAR
    ii : Pointpointer;
  BEGIN
     mark(p);
     LEES_PTS_TRIS(naam);
     ii := firstpoint;
     REPEAT
       ii^.Fpt^ := f_geb_2(beta, aaa, ii^.x, ii^.y);
       ii := ii^.Ppt;
     UNTIL ii=nil;
     Integ_Geb_2 := Double_Integral;
     release(p);
  END;

  END.
```