

AUTOMATING WELDED GUSSET PLATE CONNECTIONS
IN PLANAR TRUSS STRUCTURES

Benjamin Linder

Department of Mechanical Engineering
and Applied Mechanics

The University of Michigan

September, 1991
ME490 Report

FORWARD

The Integrated Structural Optimization System (ISOS) is a computing environment for the overall design of planar structures and is described in (Papalambros and Chirehdast 1990, Bremicker et al. 1991). The design process implemented by ISOS is carried out in four phases. For truss structures, phases I-III automatically generate a size-optimized topology with little aid from the designer. At this point however, the truss design still needs considerable refinement before it can be fabricated. The designer must determine the member cross-sections, the member layout, and the individual joint geometries. Phase IV embodies this refinement stage of a truss structure's design. Automatic generation of truss connections using gusset-plates and weldments is outlined in (Chirehdast et al. 1991) as one example of a phase IV activity. A computer program was written to automate a portion of this process. This paper establishes the progress made in phase IV for truss refinement and serves as a reference for further work.

SUMMARY

A computer program was developed as a first step towards implementation of the ideas set forth in (Chirehdast et al. 1991) for the refinement of truss structures. Specifically, the program automatically generates the weld geometry and the gusset plate geometry for each joint of a truss. The program's procedures were written anticipating the future integration of an optimization package. Important extensions of the program would be implementing methods for simplifying the resulting gusset plate and implementing methods to take advantage of a structure's symmetry.

INTRODUCTION

This paper describes a PASCAL computer program that is an example of automated truss refinement in phase IV of the Integrated Structural Optimization System (ISOS). The program automatically generates information necessary for the fabrication of welded truss connections using gusset plates. The following steps are necessary for the design of welded joints using gusset plates: (i) selection of member cross-sections; (ii) selection of member layout, (iii) calculation of weld geometry; (iv) determination of gusset plate geometry. The process for automating each of these steps is outlined in (Chirehdast et al. 1991). Steps (iii) and (iv) are automated by the computer program, step (ii) is partially automated, and step (i) was not automated.

This paper gives example output, describes the programs operation, and outlines important extensions to the program for future work. Familiarity with the treatment of Phase IV of ISOS described in (Chirehdast et al. 1991) is assumed for the following discussion.

1.0 EXAMPLE OUTPUT

Figure 1 shows the program's output for an eight bar truss obtained from phases I-III of ISOS. The gusset plates generated for each joint are shown in bold. Detailed views of the three- and four-member joints of the eight bar truss are shown in Fig. 2. The subpolygons for each member of the joint which allow for weld and clearance area are also shown in this figure. Figure 3 is an example that illustrates the importance of unconnected leg orientation. The gusset plates for the three-member joints differ greatly in size due to differences in orientation of the joint's members. The program does not automatically orient the unconnected leg. Figures 1, 2 and 3 are actual graphical output from the program. The subpolygons were removed from Fig. 1 and Fig. 3 for clarity.

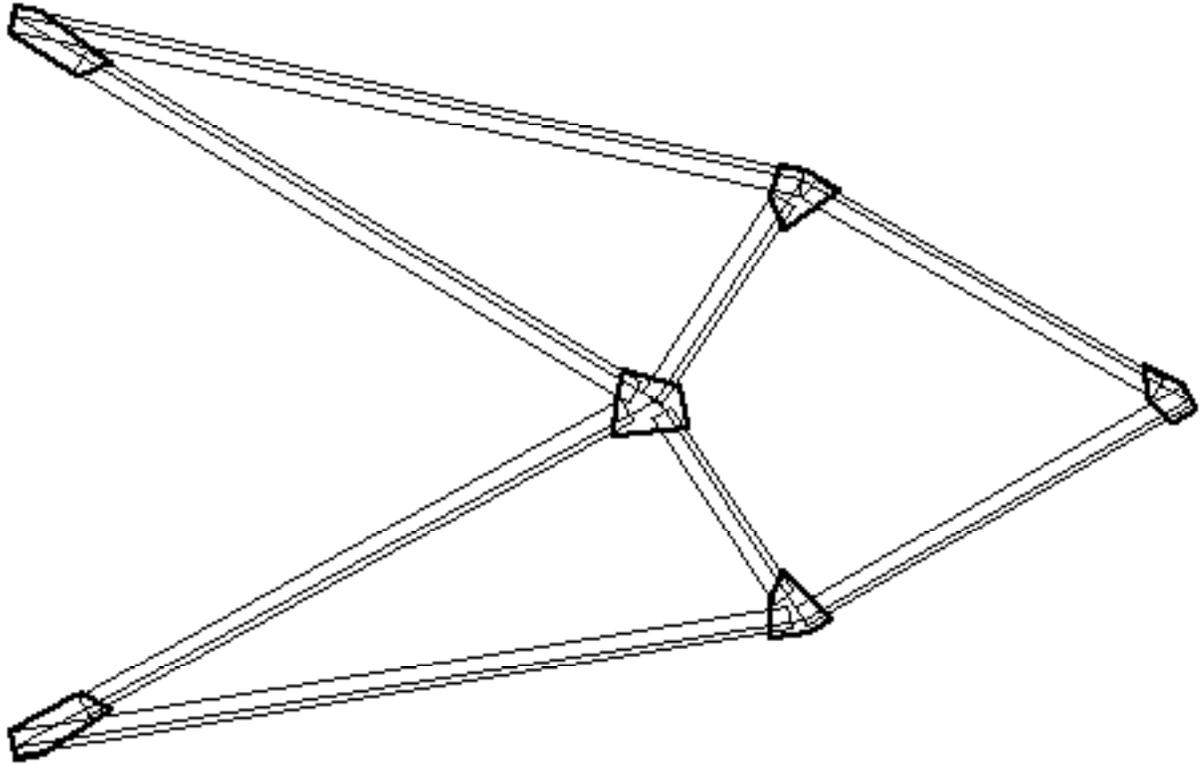


Figure 1. Program output for an eight bar truss. The truss members have their neutral axis aligned to the nodal graph. The gusset plates for each joint are shown in bold.

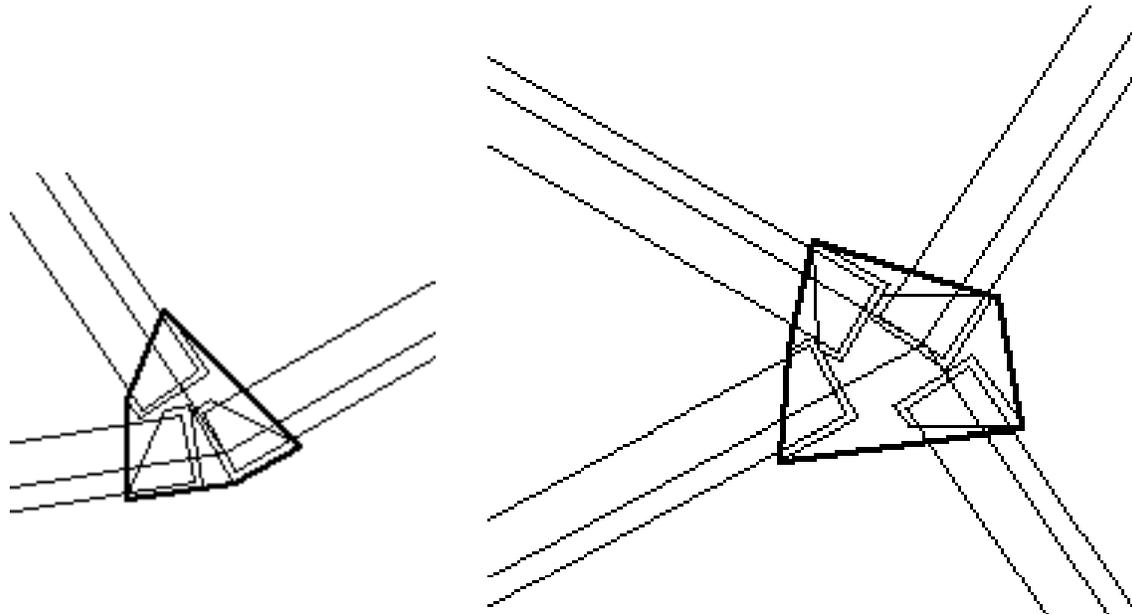


Figure 2. Exploded views of the three- and four-member joints of Fig. 1. The subpolygons shown depict the weld layout for each member. The gusset plates are shown in bold.

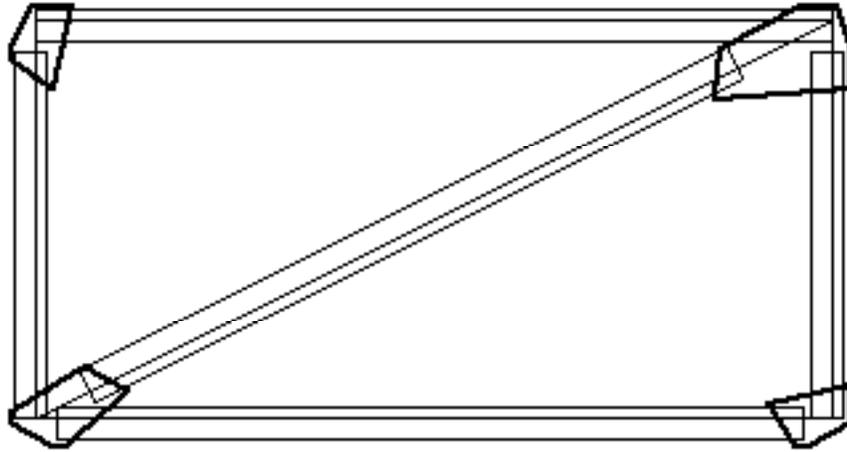


Figure 3. The importance of the orientation of the unconnected leg on the gusset plate size is illustrated by the difference in plate size for the three-member joints.

2.0 PROGRAM OPERATION

The automation of welded truss joints using gusset plates presents itself as an optimization problem. Several parameters influence the characteristics of the final design, specifically the size of the gusset plate. These include the size and thickness of the connected leg, orientation of the unconnected leg, size of the weld, and order of member cropping. One of the overall goals of ISOS is to generate a minimum weight structure. For this reason, the program was written so that the gusset plate area is kept to a minimum. For example, the maximum allowable weld size is used to minimize the weld length necessary for each member. This, in turn, reduces the required plate size. The program and the examples given here do not involve the use of an optimization package. The program's functions and procedures, however, were written with the intention that an optimizer would be called to minimize the gusset plate area. The work

to incorporate an optimizer was begun by the author in conjunction with Doctoral Candidate Mehran Chirehdast and was not completed at this time.

The program requires that the following information is given: nodal graph, member cross-section, member load, and member cropping order. A single angle cross-section and a combination end- and side-fillet weld topology are assumed. The program generates the weld geometry, crops the members, and generates the gusset plate geometry.

Figure 4 gives a flow diagram of the steps carried out by the program. Each of the main steps is described in the following subsections. Further details of how the program implements the methods discussed can be found in the comments of the program code included in the appendix. Several of the procedures in this program were developed based on concepts presented in (Pavlidis 1982).

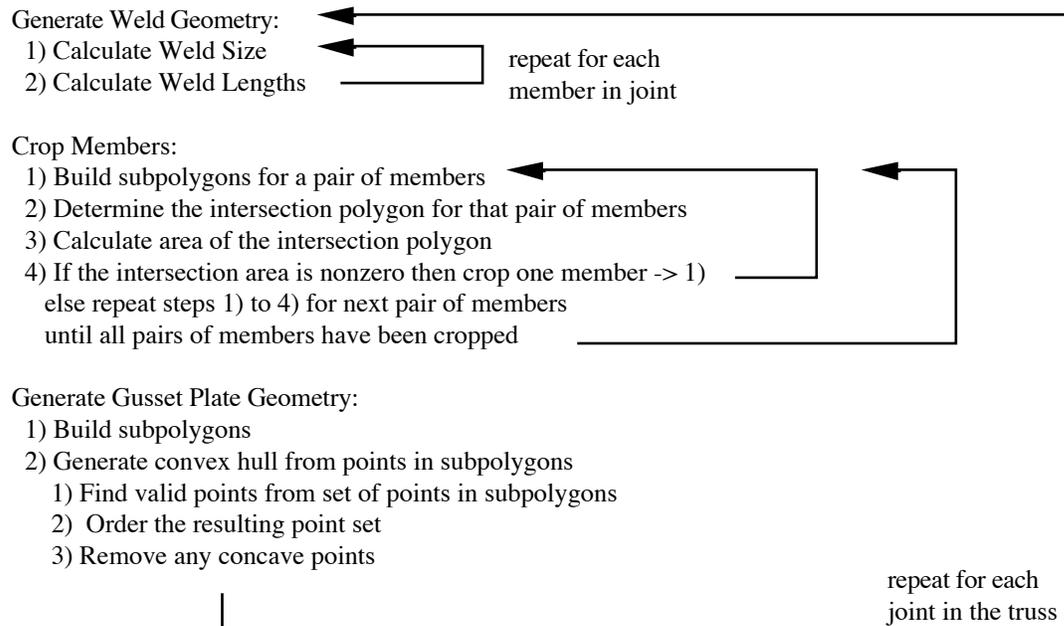


Figure 4. Flow diagram showing the steps involved for finding the weld geometry, cropping the members, and finding the gusset plate geometry.

2.1 Generating the Weld Geometry

Two steps are carried out for the generation of the weld geometry: (i) calculation of the weld size; (ii) calculation of the weld lengths. The weld size is used to calculate the weld lengths. The weld lengths are used to generate the subpolygons for the end of each member at a joint. The program assumes a combination end- and side-fillet weld topology. This type of weld requires less gusset plate area than a side-fillet weld. An end-fillet weld alone usually does not provide enough weld length. The weld size calculated is the maximum allowable weld size given by standards for a given connected leg thickness. Larger weld sizes reduce the required length of weld which in turn reduces the gusset plate size. The calculations for weld lengths are identical to the example given in (Chirehdast et al. 1991) for a single angle member welded to a gusset plate.

2.2 Cropping the Members

The cropping of a member requires three basic steps: (i) calculation of the intersection area with another member; (ii) checking if the intersection area is non-zero; (iii) reducing the member length by a fixed amount. These steps are repeated until the intersection area is zero. The procedure is carried out for all combinations of members at a joint until no two members intersect. The order in which the members are cropped is the order in which the members are listed in the connected member list stored for each node. The necessary reduction in member length could be calculated exactly using geometry; however, such a method does not lend itself to the use of an optimization package. This method allows the member to be adjusted per instructions from an optimizer.

The intersection area is obtained by first generating subpolygons that enclose the weld geometry and then creating polygons for the intersections between two member

polygons or their subpolygons. The areas of these intersection polygons are then calculated and summed.

2.3 Generating the Gusset Plate Geometry

The generation of the gusset plate for a joint takes two basic steps: (i) generating the subpolygons for all the members of a joint; (ii) generating the convex hull of the set of points given by the subpolygons. This method provides area for the weld patterns of each member, and each of these areas is smoothly connected. Unless one member is left uncropped, however, special cases exist where this method may not guarantee plate material at the nodal coordinate for a joint. An alternate subpolygon was tried that included the weld geometry but also extended to the nodal coordinate of the joint. This method guarantees material to the nodal coordinate for all members; however, it often generates excess plate material that is not needed. Another approach, which was not tried, would include the points of a polygon added at the nodal coordinate in the set of points used to generate the convex hull. This additional polygon is usually needed for joints that involve connections to supports.

3.0 EXTENSIONS TO THE PROGRAM

The program represents preliminary work in automatic truss refinement for phase IV of ISOS. Three important areas exist for further development and are described in the following subsections. These three areas are layout automation, gusset plate simplification, and symmetry incorporation.

3.1 Layout Automation

The program will align the members' neutral axis to the nodal graph and crop the members in the given order. The program does not determine the crop sequence or the

orientation of the unconnected leg. Figure 3 illustrates the importance of the unconnected leg's orientation. These two problems can be solved by incorporating an optimization package. The optimizer would take the crop distance and member orientation as variables and the area of the gusset plate as an objective function. The conditions for a solution are a minimum gusset plate area with no members intersecting.

3.2 Gusset Plate Simplification

The gusset plates generated are usually undesirable shapes to fabricate. The joints in Fig. 2, for example, have gusset plates with five sides. Simplification of the gusset plate results in a plate that is realistic to fabricate but at the expense of adding plate area. The program does not make any simplification of the gusset plate. Merging adjoining sides that are nearly collinear would greatly simplify the plates generated. In Fig. 2 for example, the plate for the three-member joint would simplify to three sides and the plate for the four-member joint would simplify to four sides. Also, (Chirehdast et al. 1991) presents discussion on the possibility of fitting simpler geometric shapes to the gusset plate. The plates in Fig. 2 could be reduced to a triangle for the three-member joint and a square for the four-member plate.

3.3 Symmetry Incorporation

The cost of fabricating a truss can be reduced by maintaining symmetry of the structure. Also, by recognizing symmetry the amount of computation during analysis can be reduced. For the truss of Fig. 1, two pairs of the joints are symmetric. Thus, only four of the six joints must be analyzed to generate all six gusset plates. Notice that even though the members of the truss are laid out symmetrically, the two center-line joints did not remain symmetric. This symmetry can be maintained by variable linking of the crop distances for the appropriate members. Symmetry methods described in (Chirehdast et

al. 1991) for phase II of ISOS are directly applicable here for automatically determining which members need variable linking.

REFERENCES

- BREMICKER, B., CHIREHDAST, M., KIKUCHI, N., PAPALAMBROS, P., 1991, Integrated Topology and Shape Optimization in Structural Design, *Journal of Mechanics of Machines and Structures*, to be published.
- CHIREHDAST, M., LINDER, B., YANG, J., PAPALAMBROS, P., 1991, Concurrent Engineering in Optimal Structural Design, to be published.
- PAPALAMBROS, P., CHIREHDAST, M., 1990, An Integrated Environment for Structural Configuration Design, *Journal of Engineering Design*, Vol. 1, No. 1, pp. 73-96.
- PAVLIDIS, T., 1982, *Algorithms for Graphics and Image Processing*, (Computer Science Press, Rockville, MD).

APPENIX

```
{WRITTEN BY BENJAMIN LINDER }
{7/91 }
{MANY THANKS TO MEHRAN CHIRAHDAST FOR HIS HELP }
```

```
program GussetPlate;
```

```
uses
  DataStructure, Drawing, Utilities, WeldGeometry, LayoutGeometry, GussetPlateGeometry, Examples,
  Optimize;
```

```
var
  area, PlateArea: double;
  Poly, PtSet: PolygonType;
  Nnum, Mnum: integer;
  i, j, k: integer;
```

```
{*****}
{this file contains the initialize procedure and the main program}
{*****}
```

```
procedure Initialize;
{this procedure initializes parts of the data structure}
```

```
var
  i, j, k: integer;
begin
  Scale := 0.069;
  ShiftX := 20;
  ShiftY := 10;
```

```
{Scale := 0.10;}
{ShiftX := 50;}
{ShiftY := 50;}
  WeldClearance := 1.5;
  AllowWeldLoad := 0.70;
```

```
for i := 1 to NumNodes do
  for j := 1 to Node[i].NumConnected do
    begin
      Node[i].MemberEnd[j].CropDist := 0;
      Node[i].GussetPlate.Size := 0;
      for k := 1 to 5 do
        begin
          Node[i].MemberEnd[j].Poly[k].x := 0;
          Node[i].MemberEnd[j].Poly[k].y := 0;
        end;
      end;
    end;
```

```
begin
```

```

Initialize;                {initialize the data structure}

                                                                    {load one of the examples}
{HardCodeThree;}
{HardCodeFour;}
HardCodeSupport;

BuildMemberPolygons;      {build the member polygons}
DrawGraph;                {draw the nodal graph of the truss}

                                                                    {calculate the weldsizes and weld
lengths for all members}
for i := 1 to NumMembers do
begin
  Member[i].S := WeldSize(i);
  WeldLengths(i, Member[i].Weld1, Member[i].Weld2, Member[i].Weld3);
end;

                                                                    {this big set of loops takes care of
cropping the members at }
                                                                    {at each joint}

for i := 1 to numnodes do
begin
  for j := 1 to Node[i].NumConnected - 1 do
  for k := j + 1 to Node[i].NumConnected do
  begin
    BuildSubPolygons_2(i);
    Poly := OrderPtSet(IntersectionPtSet(Node[i].MemberEnd[j].poly,
Member[Node[i].MemberEnd[k].Member].Poly));
    area := Polygonarea(poly);
    Poly := OrderPtSet(IntersectionPtSet(Node[i].MemberEnd[j].poly, Node[i].MemberEnd[k].Poly));
    area := area + Polygonarea(poly);
    while area > 0 do
    begin
      Node[i].MemberEnd[j].cropdist := Node[i].MemberEnd[j].cropdist + 10;

      {**** this is the crop increment ^^^^^^^}

      {it is how much is taken off each time a member is }

{shortended by CropMemberEnd}

      CropMemberEnd(Node[i].MemberEnd[j].Member, i, Node[i].MemberEnd[j].CropDist);
      BuildSubPolygons_2(i);
      Poly := OrderPtSet(IntersectionPtSet(Node[i].MemberEnd[j].poly,
Member[Node[i].MemberEnd[k].Member].Poly));
      area := Polygonarea(poly);
      Poly := OrderPtSet(IntersectionPtSet(Node[i].MemberEnd[j].poly, Node[i].MemberEnd[k].Poly));
      area := area + Polygonarea(poly);
    end;
  end;
end;
end;

```

```

DrawMembers;                                {draw the new cropped member
polygons}

PlateArea := 0;
for Nnum := 1 to numnodes do
begin
  DrawSubPolygons(Nnum);                    {draw the sub polygons for the node}

the gusset plate by finding the pt set, ordering it, and}
removeing the concave pts giving a convex hull}
  Node[Nnum].GussetPlate := RemoveConcavePts(OrderPtSet(FindGussetPlatePts(Nnum)));

the gusset plate for the node}
  DrawPolygon(Node[Nnum].GussetPlate);

  {calculate the plate area for the heck of it}
  PlateArea := PlateArea + PolygonArea(Node[Nnum].GussetPlate);
end;

Writeln;
Writeln('Total Plate area = ', PlateArea : 0 : 0);
end.

```

```

unit DataStructure;

interface

type
  PointType = record
    x, y: Double;
  end;

  LineType = record
    a, b, c: Double;
  end;

  PolygonType = record      {used for storing the gusset plate polygon}
    size: integer;
    pts: array[1..50] of PointType;
  end;

  SubPolygonType = array[1..5] of PointType;

  CrossSectionType = record {cross-section data for each member}
    Area: Double;
    t, y, I1, I2: Double;
  end;

  MemberType = record
    Node1, Node2: integer;
    Poly: SubPolygonType;
    Load: Double;
    CrossSection: CrossSectionType;
    Weld1, Weld2, Weld3: Double;
    S: Double;
  end;

  MemberEndType = record      {the info stored in each entry in the connected member list IN each node}
    Member: integer;
    Poly: SubPolygonType;
    CropDist: Double;
  end;

  NodeType = record
    location: PointType;
    NumConnected: integer;
    MemberEnd: array[1..10] of MemberEndType;
    GussetPlate: PolygonType;
  end;

  IntersectionType = (MidPoint, Line, Empty, EndPoint, TotalOverlap, PointOverlap, PartialOverlap,
  EmptyOverlap);

  XType = array[1..100] of double;      {will be used by the optimizer}
  Gtype = array[1..100] of double;

var
  {Node contains a list of all the nodes in a structure. For each node its location, number of members }
  {connected to it, a list containing info on the members connected, and the gusset plate are stored with}

```

{each node}

Node: array[1..10] of Nodetype;

{Member contains a list of all the members in the structure. The purpose of this list is to store info}
{about each member such as the nodes it connects, cross-section data, load and weld data. Weld}
{data can be stored in the member because both ends of the member have the same weld pattern}
{because they carry the same load}

Member: array[1..20] of MemberType;

{NumNodes and NumMembers are the Number of Nodes and Members in a truss}

NumNodes: integer;

NumMembers: integer;

{CurrentNode will be used by the optimizer to store the current node}

CurrentNode: integer;

{Scale, ShiftX, ShiftY are used by the drawing routines these are tweaked by me to get a truss}
{that is way to big for the screen scaled down so it does fit on the screen}

Scale, ShiftX, ShiftY: Double;

{WeldClearance is the clearance given for each weld which is 1.5*weldsize for a given member }
{so that there is room for the weld plus some clearance. AllowWeldLoad is the allowable stress}
{per unit length of weld which is specified by an electrode type}

WeldClearance, AllowWeldLoad: Double;

implementation

end.

```

unit Drawing;

interface
uses
  DataStructure;

procedure DrawPolygon (Poly: PolygonType);
procedure DrawSubPolygons (NodeNum: integer);
procedure DrawGraph;
procedure DrawMembers;

implementation

{*** }
{ this unit contains all of the routines called to draw things to the screen. You will need}
{to resize the drawing window that comes up the first time you run the program so that its }
{big enough to fit the picture in}
{*** }

procedure DrawPolygon (Poly: PolygonType); {draws the gusset plate polygon}
var
  i: integer;
begin
  showdrawing;
  PenSize(2, 2);
  moveto(round((Poly.pts[1].x - 1) * Scale + ShiftX), round((Poly.pts[1].y - 1) * Scale + ShiftY));
  for i := 2 to Poly.size do
    lineto(round((Poly.pts[i].x - 1) * Scale + ShiftX), round((Poly.pts[i].y - 1) * Scale + ShiftY));
  PenNormal;
end;

procedure DrawSubPolygons (NodeNum: integer); {draws subpolygons at the member ends}
var
  i, j: integer;
begin
  ShowDrawing;
  PenNormal;
  for i := 1 to Node[NodeNum].NumConnected do
    begin
      Moveto(Round(Node[NodeNum].MemberEnd[i].Poly[1].x * Scale + ShiftX),
Round(Node[NodeNum].MemberEnd[i].Poly[1].y * Scale + ShiftY));
      for j := 2 to 5 do
        Lineto(Round(Node[NodeNum].MemberEnd[i].Poly[j].x * Scale + ShiftX),
Round(Node[NodeNum].MemberEnd[i].Poly[j].y * Scale + ShiftY));
      end;
    end;
end;

procedure DrawGraph; {draws the nodal graph of a structure obtained from phase II or III}
var
  i, j: integer;
begin
  ShowDrawing;
  PenNormal;
  for i := 1 to NumMembers do
    begin

```

```
    Moveto(Round(Node[Member[i].Node1].Location.x * Scale + ShiftX),
Round(Node[Member[i].Node1].Location.y * Scale + ShiftY));
    Lineto(Round(Node[Member[i].Node2].Location.x * Scale + ShiftX),
Round(Node[Member[i].Node2].Location.y * Scale + ShiftY));
end;
end;
```

```
procedure DrawMembers; {draws the polygon that represents a member}
```

```
var
  i, j: integer;
begin
  ShowDrawing;
  PenNormal;
  for i := 1 to NumMembers do
  begin
    Moveto(Round(Member[i].Poly[1].x * Scale + ShiftX), Round(Member[i].Poly[1].y * Scale + ShiftY));
    for j := 2 to 5 do
      Lineto(Round(Member[i].Poly[j].x * Scale + ShiftX), Round(Member[i].Poly[j].y * Scale + ShiftY));
    end;
  end;
end.
```

unit Utilities;

interface

uses

DataStructure, Drawing;

function Distance (pt1, pt2: PointType): Double;
function angle (startpt, endpt: PointType): Double;
function IncludedAngle (CenterPt, Pt1, Pt2: PointType): Double;
function tan (angl: Double): Double;
function PtOutsidePolygon (P: PointType; Poly: SubPolygonType): Boolean;
function PtOnPolygon (P: PointType; Poly: SubPolygonType): Boolean;
function PtInsidePolygon (P: PointType; Poly: SubPolygonType): Boolean;
function PolygonArea (Poly: PolygonType): Double;
function OrderPtSet (PtSet: PolygonType): PolygonType;
function RemoveConcavePts (Poly: PolygonType): PolygonType;

implementation

```
{***}  
{This unit contains all of the functions that are considered utilities, which means they are generic}  
{to some extent.}  
{***}
```

```
function Distance (pt1, pt2: PointType): Double; {calc the distance between two pts}  
var  
  d1, d2: Double;  
begin  
  d1 := pt2.x - pt1.x;  
  d2 := pt2.y - pt1.y;  
  Distance := SQRT(d1 * d1 + d2 * d2);  
end;
```

```
function angle (startpt, endpt: PointType): Double; {calc angle between a line described by }
```

```
  {two pts and the x-axis in DEGREES}
```

```
var  
  a, b: Double;  
begin  
  a := endpt.x - startpt.x;  
  b := endpt.y - startpt.y;  
  if (b = 0) and (a > 0) then  
    angle := 0  
  else if (b = 0) and (a < 0) then  
    angle := 180  
  else if (a = 0) and (b > 0) then  
    angle := 90  
  else if (a = 0) and (b < 0) then  
    angle := 270  
  else if (a > 0) and (b >= 0) then  
    angle := arctan(b / a) * 180 / pi  
  else if (a > 0) and (b < 0) then  
    angle := arctan(b / a) * 180 / pi + 360  
  else if (a < 0) and (b >= 0) then
```

```

    angle := 180 + arctan(b / a) * 180 / pi
else if (a < 0) and (b < 0) then
    angle := 180 + arctan(b / a) * 180 / pi;
end;

```

```

function IncludedAngle (CenterPt, Pt1, Pt2: PointType): Double; {included angle given by three}

```

```

                                     {ordered pts that represent two}

```

```

                                     {lines that intersect in a pt}

```

```

var
    a, a1, a2: Double;
begin
    a1 := angle(CenterPt, Pt1);
    a2 := angle(CenterPt, Pt2);
    if a1 > a2 then
        a := a2 + (360 - a1)
    else if a2 >= a1 then
        a := a2 - a1;
    IncludedAngle := a;
end;

```

```

function tan (angl: Double): Double;
begin
    tan := sin(angl * pi / 180) / cos(angl);
end;

```

```

function PtOutsidePolygon (P: PointType; Poly: SubPolygonType): Boolean;

```

```

                                     {returns true if a pt}

```

```

                                     { is strictly outside the polygon}

```

```

var
    i: integer;
    a, S, x1, x2, y1, y2: Double;
begin
    a := 0;
    for i := 1 to 4 do
        begin
            x1 := Poly[i].x;
            y1 := Poly[i].y;
            x2 := Poly[i + 1].x;
            y2 := Poly[i + 1].y;
            S := P.x * (y1 - y2) + P.y * (x2 - x1) + (x1 * y2 - y1 * x2);
            if S > 0 then
                a := 1
            else if (S = 0) and ((x1 <> x2) and (((P.x > x1) and (P.x > x2)) or ((P.x < x1) and (P.x < x2)))) then
                a := 1
            else if (S = 0) and ((x1 = x2) and (((P.y > y1) and (P.y > y2)) or ((P.y < y1) and (P.y < y2)))) then
                a := 1;
            end;
        end;
    if a <> 0 then
        PtOutsidePolygon := true
    else

```

```
PtOutsidePolygon := false;
end;
```

```
function PtOnPolygon (P: PointType; Poly: SubPolygonType): Boolean;
```

```

{
    {returns true if the pt lies on}
    {the polygon's edge}
}
var
i: integer;
a, S, x1, x2, y1, y2: Double;
begin
a := 0;
for i := 1 to 4 do
begin
x1 := Poly[i].x;
y1 := Poly[i].y;
x2 := Poly[i + 1].x;
y2 := Poly[i + 1].y;
S := P.x * (y1 - y2) + P.y * (x2 - x1) + (x1 * y2 - y1 * x2);
if (S = 0) and ((x1 <> x2) and (((P.x <= x1) and (P.x >= x2)) or ((P.x >= x1) and (P.x <= x2)))) then
a := 1
else if (S = 0) and ((x1 = x2) and (((P.y <= y1) and (P.y >= y2)) or ((P.y >= y1) and (P.y <= y2)))) then
a := 1;
end;
if a <> 0 then
PtOnPolygon := true
else
PtOnPolygon := false;
end;
```

```
function PtInsidePolygon (P: PointType; Poly: SubPolygonType): Boolean;
```

```

{returns true if the pt}
{lies strictly inside the polygon}
}
var
i: integer;
a, S, x1, x2, y1, y2: Double;
x, y: integer;
begin
a := 0;
x := round(p.x);
y := round(p.y);
for i := 1 to 4 do
begin
x1 := Poly[i].x;
y1 := Poly[i].y;
x2 := Poly[i + 1].x;
y2 := Poly[i + 1].y;
S := P.x * (y1 - y2) + P.y * (x2 - x1) + (x1 * y2 - y1 * x2);
if S < 0 then
a := a + 1;
end;
```

```

if a = 4 then
  PtInsidePolygon := true
else
  PtInsidePolygon := false;
end;

```

```

function PolygonArea (Poly: PolygonType): Double; {returns the area of a polygon whose pts are}

```

```

    {are ordered clockwise}

```

```

var
  i: integer;
  area: Double;
begin
  area := 0;

  if Poly.Size = 0 then
    area := 0
  else
    begin
      for i := 1 to Poly.Size - 1 do
        area := area + (Poly.pts[i + 1].x * Poly.pts[i].y - Poly.pts[i].x * Poly.pts[i + 1].y);
      area := area / 2;
    end;
  PolygonArea := area;
end;

```

```

function OrderPtSet (PtSet: PolygonType): PolygonType; {orders a set of pts stored in Ptset}

```

```

    {according to their angle with the x-axis}

```

```

    {first step in finding a convex hull}

```

```

    {the result is a set of pts ordered clockwise}

```

```

var
  i, j, n: integer;
  TempPt, XaxisPt: PointType;
  TempAng, MaxAng, Ymin, MinDist, TempDist: Double;
begin
  if PtSet.Size < 3 then
    begin
      OrderPtSet := PtSet;
      exit(OrderPtSet);
    end;

  n := 1;
  Ymin := PtSet.Pts[1].y;
  for i := 2 to PtSet.Size do
    if PtSet.Pts[i].y < Ymin then
      begin
        n := i;
        Ymin := PtSet.Pts[i].y;
      end;
  TempPt := PtSet.Pts[1];

```

```

    {find a pt that has the smallest Y value}

```

```

PtSet.Pts[1] := PtSet.Pts[n];
PtSet.Pts[n] := TempPt;

XaxisPt.x := PtSet.Pts[1].x + 1;
XaxisPt.y := PtSet.Pts[1].y;

for i := 2 to PtSet.Size - 1 do
  {order the pts according to angle}
begin
  {if angles are same take the one}
  {closest to the pt with minimum Y}
  MaxAng := IncludedAngle(PtSet.Pts[1], XaxisPt, PtSet.Pts[i]);
  MinDist := Distance(PtSet.Pts[1], PtSet.Pts[i]);
  if MaxAng > 180 then
    MaxAng := 360 - MaxAng;
  for j := i + 1 to PtSet.Size do
    begin
      TempAng := IncludedAngle(PtSet.Pts[1], XaxisPt, PtSet.Pts[j]);
      TempDist := Distance(PtSet.Pts[1], PtSet.Pts[j]);
      if TempAng > 180 then
        TempAng := 360 - TempAng;
      if TempAng > MaxAng then
        begin
          MaxAng := TempAng;
          MinDist := TempDist;
          TempPt := PtSet.Pts[j];
          PtSet.Pts[j] := PtSet.Pts[i];
          PtSet.Pts[i] := TempPt;
        end
      else if (TempAng = MaxAng) and (TempDist < MinDist) then
        begin
          MinDist := TempDist;
          TempPt := PtSet.Pts[j];
          PtSet.Pts[j] := PtSet.Pts[i];
          PtSet.Pts[i] := TempPt;
        end;
      end;
    end;
  end;

PtSet.Size := PtSet.Size + 1;
PtSet.Pts[PtSet.Size] := PtSet.Pts[1];
OrderPtSet := PtSet;
end;

function RemoveConcavePts (Poly: PolygonType): PolygonType; {removes all points in a polygon}
  {that make it nonconvex}
  {the second step in finding a}
  {convex hull. The result is a }
  {convex polygon}

var
i, j: integer;

```

```

    a1, a2, a: Double;
begin
    i := 1;
    while (i <> Poly.Size - 1) and (Poly.Size > 3) do
    begin
        a1 := Angle(Poly.Pts[i + 1], Poly.Pts[i]);
        a2 := Angle(Poly.Pts[i + 1], Poly.Pts[i + 2]);
        if a1 >= a2 then
            a := a1 - a2
        else if a1 < a2 then
            a := a1 + (360 - a2);

        if a < 180 then
        begin
            for j := i + 1 to Poly.Size - 1 do    {remove the non-concave pt}
                Poly.Pts[j] := Poly.Pts[j + 1];
            Poly.Size := Poly.Size - 1;
            Poly := RemoveConcavePts(Poly);    {recursive calls until the non concave pts}

                { are all gone}
        leave;
        end
        else
            i := i + 1;
        end;

        RemoveConcavePts := Poly;
    end;

end.

```

```

unit SegmentIntersection;

interface
uses
  DataStructure;

procedure SegmentIntersection (Pt1, Pt2, Pt3, Pt4: PointType; var Intersection: IntersectionType; var
InterPoint: PointType);

implementation

{***}
{this unit is just for segment intersection. Given two segments it will find out how they intersecta}
{and return it in Intersection and if they intersect in a pt it will find that pt. The segments are}
{defined by their endpts}
{***}

function LineFromPoints (Pt1, Pt2: PointType): LineType;
begin
  LineFromPoints.a := Pt1.y - Pt2.y;
  LineFromPoints.b := Pt2.x - Pt1.x;
  LineFromPoints.c := Pt1.x * Pt2.y - Pt1.y * Pt2.x;
end;

function LineIntersection (Line1, Line2: LineType): PointType;
var
  x, y, w: Double;
begin
  x := Line1.b * Line2.c - Line1.c * Line2.b;
  y := Line1.c * Line2.a - Line1.a * Line2.c;
  w := Line1.a * Line2.b - Line1.b * Line2.a;
  if w <> 0 then
    begin
      LineIntersection.x := x / w;
      LineIntersection.y := y / w;
    end
  else
    begin
      LineIntersection.x := x;
      LineIntersection.y := y;
    end;
end;

procedure SegmentOverlap (Pt1, Pt2, Pt3, Pt4: PointType; var Intersection: IntersectionType; var
InterPoint: PointType);
var
  x1, x2, x3, x4, y1, y2, y3, y4: Double;
begin
  x1 := Pt1.x;
  x2 := Pt2.x;
  x3 := Pt3.x;
  x4 := Pt4.x;
  y1 := Pt1.y;
  y2 := Pt2.y;

```

```

y3 := Pt3.y;
y4 := Pt4.y;

if x1 = x2 then
begin
if ((y1 >= y3) and (y1 <= y4) and (y2 >= y3) and (y2 <= y4)) then
Intersection := TotalOverlap
else if ((y1 >= y4) and (y1 <= y3) and (y2 >= y4) and (y2 <= y3)) then
Intersection := TotalOverlap
else if ((y1 > y3) and (y1 > y4) and (y2 > y3) and (y2 > y4)) then
Intersection := EmptyOverlap
else if ((y1 < y3) and (y1 < y4) and (y2 < y3) and (y2 < y4)) then
Intersection := EmptyOverlap
else if (y2 = y3) and (x2 = x3) and (((y1 < y2) and (y3 < y4)) or ((y2 < y1) and (y4 < y3))) then
begin
Intersection := PointOverlap;
InterPoint := Pt2;
end
else if (y2 = y4) and (x2 = x4) and (((y1 < y2) and (y4 < y3)) or ((y2 < y1) and (y3 < y4))) then
begin
Intersection := PointOverlap;
InterPoint := Pt2;
end
else if (y1 = y3) and (x1 = x3) and (((y2 < y1) and (y3 < y4)) or ((y1 < y2) and (y4 < y3))) then
begin
Intersection := PointOverlap;
InterPoint := Pt1;
end
else if (y1 = y4) and (x1 = x4) and (((y2 < y1) and (y4 < y3)) or ((y1 < y2) and (y3 < y4))) then
begin
Intersection := PointOverlap;
InterPoint := Pt1;
end
else
Intersection := PartialOverlap;
end
else if x1 <> x2 then
begin
if ((x1 >= x3) and (x1 <= x4) and (x2 >= x3) and (x2 <= x4)) then
Intersection := TotalOverlap
else if ((x1 >= x4) and (x1 <= x3) and (x2 >= x4) and (x2 <= x3)) then
Intersection := TotalOverlap
else if ((x1 > x3) and (x1 > x4) and (x2 > x3) and (x2 > x4)) then
Intersection := EmptyOverlap
else if ((x1 < x3) and (x1 < x4) and (x2 < x3) and (x2 < x4)) then
Intersection := EmptyOverlap
else if (x2 = x3) and (y2 = y3) and (((x1 < x2) and (x3 < x4)) or ((x2 < x1) and (x4 < x3))) then
begin
Intersection := PointOverlap;
InterPoint := Pt2;
end
else if (x2 = x4) and (y2 = y4) and (((x1 < x2) and (x4 < x3)) or ((x2 < x1) and (x3 < x4))) then
begin
Intersection := PointOverlap;
InterPoint := Pt2;
end
end
end

```

```

else if (x1 = x3) and (y1 = y3) and (((x2 < x1) and (x3 < x4)) or ((x1 < x2) and (x4 < x3))) then
begin
  Intersection := PointOverlap;
  InterPoint := Pt1;
end
else if (x1 = x4) and (y1 = y4) and (((x2 < x1) and (x4 < x3)) or ((x1 < x2) and (x3 < x4))) then
begin
  Intersection := PointOverlap;
  InterPoint := Pt1;
end
else
  Intersection := PartialOverlap;
end;
end;

```

```

procedure SegmentIntersection (Pt1, Pt2, Pt3, Pt4: PointType; var Intersection: IntersectionType; var
InterPoint: PointType);

```

```

var
  S1, S2, S3, S4: Double;
  x1, x2, x3, x4, y1, y2, y3, y4: Double;
  L12, L34: LineType;

begin
  {uses the method of signs to determine intersection}
  x1 := Pt1.x;
  x2 := Pt2.x;
  x3 := Pt3.x;
  x4 := Pt4.x;
  y1 := Pt1.y;
  y2 := Pt2.y;
  y3 := Pt3.y;
  y4 := Pt4.y;

  S1 := x1 * (y3 - y4) + y1 * (x4 - x3) + (x3 * y4 - y3 * x4);
  S2 := x2 * (y3 - y4) + y2 * (x4 - x3) + (x3 * y4 - y3 * x4);
  S3 := x3 * (y1 - y2) + y3 * (x2 - x1) + (x1 * y2 - y1 * x2);
  S4 := x4 * (y1 - y2) + y4 * (x2 - x1) + (x1 * y2 - y1 * x2);

  if (S1 * S2 < 0) and (S3 * S4 < 0) then
begin
  Intersection := MidPoint;
  L12 := LineFromPoints(Pt1, Pt2);
  L34 := LineFromPoints(Pt3, Pt4);
  InterPoint := LineIntersection(L12, L34);
end
else if ((S1 * S2 > 0) and (S3 * S4 <= 0)) or ((S1 * S2 <= 0) and (S3 * S4 > 0)) then
  Intersection := Empty
else if (S1 * S2 > 0) and (S1 * S2 > 0) then
  Intersection := Empty
else if (((S1 = 0) and (S2 <> 0)) and ((S3 * S4 < 0) or (S3 = 0) or (S4 = 0))) then
begin
  Intersection := EndPoint;
  InterPoint := Pt1;
end
else if (((S2 = 0) and (S1 <> 0)) and ((S3 * S4 < 0) or (S3 = 0) or (S4 = 0))) then

```

```
begin
  Intersection := EndPoint;
  InterPoint := Pt2;
end
else if ((S3 = 0) and (S4 <> 0)) and (S1 * S2 < 0) then
begin
  Intersection := EndPoint;
  InterPoint := Pt3;
end
else if ((S4 = 0) and (S3 <> 0)) and (S1 * S2 < 0) then
begin
  Intersection := EndPoint;
  InterPoint := Pt4;
end
else if ((S1 = 0) and (S2 = 0) and (S3 = 0) or (S4 = 0)) then
  SegmentOverlap(Pt1, Pt2, Pt3, Pt4, Intersection, InterPoint)
else
  Intersection := Empty;
end;
end.
```

```

unit GussetPlateGeometry;

interface

uses
  DataStructure, Utilities;

function FindGussetPlatePts (Nnum: integer): PolygonType;

implementation

{****}
{this unit contains the function for finding the ptset that a convex hull will be formed out of to get}
{the gusset plate polygon}
{***}

{this function finds the valid pts from the set of pts given by the pts in each of the subpolygons}
{some pts lie inside other sub polygons so they can not be part of the convex}
{hull. This function is not really necessary the convex hull generator would still get it right if}
{all of the vertices of the subpolygons were used}

function FindGussetPlatePts (Nnum: integer): PolygonType;
var
  i, j, k, l, Size: integer;
  PtSet: PolygonType;
begin
  Size := 0;
  for i := 1 to Node[Nnum].NumConnected do
    for k := 1 to 4 do
      begin
        l := 0;
        for j := 1 to Node[Nnum].NumConnected do
          begin
            if j = i then
              cycle;
            if not PtOutSidePolygon(Node[Nnum].MemberEnd[i].Poly[k], Node[Nnum].MemberEnd[j].Poly) then
              begin
                l := 1;
                leave;
              end;
            end;
            if l = 0 then
              begin
                Size := Size + 1;
                PtSet.Pts[Size] := Node[Nnum].MemberEnd[i].Poly[k];
              end;
            end;
          end;
        end;
      end;
    end;

  PtSet.Size := Size;
  FindGussetPlatePts := PtSet;
end;

end.

```

```

unit WeldGeometry;

interface
uses
  DataStructure, Utilities;

function WeldSize (Mnum: integer): Double;
procedure WeldLengths (Mnum: integer; var W1, W2, W3: Double);

implementation
{****}
{this unit contains the procedures that find the weld size and length. they implement the}
{example given in the chapter myself, julie and mehi wrote with prof Papalambros}
{****}

function WeldSize (Mnum: integer): Double;
var
  t: Double;
begin
  t := Member[Mnum].CrossSection.t;
  if t < 6.5 then
    WeldSize := t
  else if t >= 6.5 then
    WeldSize := t - 1.5;
end;

procedure WeldLengths (Mnum: integer; var W1, W2, W3: Double);
var
  L1, F1, N, y: Double;
begin
  L1 := Member[Mnum].CrossSection.L1;
  y := Member[Mnum].CrossSection.y;
  N := Member[Mnum].Load;

  W1 := L1;
  F1 := L1 * AllowWeldLoad;
  W3 := (N * y - F1 * (L1 / 2)) / ((AllowWeldLoad) * L1);
  W2 := (N - F1 - W3 * (AllowWeldLoad)) / (AllowWeldLoad);
end;

end.

```

```

unit LayoutGeometry;

interface
uses
  DataStructure, Utilities;

procedure CropMemberEnd (MNum, Nnum: integer; CropDistance: Double);
procedure BuildSubPolygons_1 (NodeNum: integer);
procedure BuildSubPolygons_2 (NodeNum: integer);
procedure BuildMemberPolygons;

implementation

{****}
{this unit calculates subpolygons, member polygons, and takes care of cropping member ends}
{****}

procedure CropMemberEnd (MNum, Nnum: integer; CropDistance: Double);
{given a member, which end to crop (given by Nnum), and a cropdistance the member polygon}
{is cropped back to the new distance}
var
  i: integer;
  ang, a, b: Double;
  EndPt: PointType;
begin
  a := Member[MNum].CrossSection.y;
  b := Member[MNum].CrossSection.l1 - a;

  if (Member[MNum].Node1 = Nnum) then
    begin
      ang := angle(Node[Nnum].Location, Node[Member[MNum].Node2].Location);
      ang := ang * pi / 180;
      Member[MNum].Poly[1].x := Node[Nnum].Location.x + a * Cos(ang - pi / 2) + CropDistance *
      Cos(ang);
      Member[MNum].Poly[1].y := Node[Nnum].Location.y + a * Sin(ang - pi / 2) + CropDistance *
      Sin(ang);
      Member[MNum].Poly[2].x := Node[Nnum].Location.x - b * Cos(ang - pi / 2) + CropDistance *
      Cos(ang);
      Member[MNum].Poly[2].y := Node[Nnum].Location.y - b * Sin(ang - pi / 2) + CropDistance *
      Sin(ang);
      Member[MNum].Poly[5] := Member[MNum].Poly[1];
    end
  else
    begin
      ang := angle(Node[Nnum].Location, Node[Member[MNum].Node1].Location);
      ang := ang * pi / 180;
      Member[MNum].Poly[3].x := Node[Nnum].Location.x + b * Cos(ang - pi / 2) + CropDistance *
      Cos(ang);
      Member[MNum].Poly[3].y := Node[Nnum].Location.y + b * Sin(ang - pi / 2) + CropDistance *
      Sin(ang);
      Member[MNum].Poly[4].x := Node[Nnum].Location.x - a * Cos(ang - pi / 2) + CropDistance *
      Cos(ang);
      Member[MNum].Poly[4].y := Node[Nnum].Location.y - a * Sin(ang - pi / 2) + CropDistance *
      Sin(ang);
    end;
  end;
end;

```

```

procedure BuildSubPolygons_1 (NodeNum: integer);
{this is the old method of building sub polygons. the polygon goes through the nodal coordinate to
guarantee}
{material to the node's location. this procedure is currently not called and the cropping code in the main}
{program would have to be altered to be able to use it.}
var
i: integer;
a, b, ang, dx, dy: Double;
Wa, Wb, Clearance: Double;
EndPt: PointType;
begin
for i := 1 to Node[NodeNum].NumConnected do
begin
Clearance := WeldClearance * Member[Node[NodeNum].MemberEnd[i].member].CrossSection.t;

if (Member[Node[NodeNum].MemberEnd[i].Member].Node1 = NodeNum) then
begin
EndPt := Node[Member[Node[NodeNum].MemberEnd[i].Member].Node2].Location;
a := Member[Node[NodeNum].MemberEnd[i].member].CrossSection.y;
b := Member[Node[NodeNum].MemberEnd[i].member].CrossSection.l1 - a;
Wa := Member[Node[NodeNum].MemberEnd[i].member].Weld2;
Wb := Member[Node[NodeNum].MemberEnd[i].member].Weld3;
end
else
begin
EndPt := Node[Member[Node[NodeNum].MemberEnd[i].Member].Node1].Location;
b := Member[Node[NodeNum].MemberEnd[i].member].CrossSection.y;
a := Member[Node[NodeNum].MemberEnd[i].member].CrossSection.l1 - b;
Wa := Member[Node[NodeNum].MemberEnd[i].member].Weld3;
Wb := Member[Node[NodeNum].MemberEnd[i].member].Weld2;
end;

ang := angle(Node[NodeNum].Location, EndPt);
ang := ang * pi / 180;
dx := (Node[NodeNum].MemberEnd[i].CropDist - Clearance) * Cos(ang);
dy := (Node[NodeNum].MemberEnd[i].CropDist - Clearance) * Sin(ang);
a := a + Clearance;
b := b + Clearance;

Node[NodeNum].MemberEnd[i].Poly[1].x := Node[NodeNum].Location.x + a * cos(ang - pi / 2);
Node[NodeNum].MemberEnd[i].Poly[1].y := Node[NodeNum].Location.y + a * sin(ang - pi / 2);

Node[NodeNum].MemberEnd[i].Poly[2].x := Node[NodeNum].Location.x - b * cos(ang - pi / 2);
Node[NodeNum].MemberEnd[i].Poly[2].y := Node[NodeNum].Location.y - b * sin(ang - pi / 2);

Node[NodeNum].MemberEnd[i].Poly[3].x := Node[NodeNum].Location.x + dx - b * cos(ang - pi / 2) +
Wb * Cos(ang);
Node[NodeNum].MemberEnd[i].Poly[3].y := Node[NodeNum].Location.y + dy - b * sin(ang - pi / 2) +
Wb * Sin(ang);

Node[NodeNum].MemberEnd[i].Poly[4].x := Node[NodeNum].Location.x + dx + a * cos(ang - pi / 2) +
Wa * Cos(ang);
Node[NodeNum].MemberEnd[i].Poly[4].y := Node[NodeNum].Location.y + dy + a * sin(ang - pi / 2) +
Wa * Sin(ang);

```

```

Node[NodeNum].MemberEnd[i].Poly[5] := Node[NodeNum].MemberEnd[i].Poly[1];
end;
end;

procedure BuildSubPolygons_2 (NodeNum: integer);
{this procedure forms a subpolygon that represents a convex hull of the weld topology including}
{clearance for the weld size and some additional clearance between welds}
var
i: integer;
a, b, ang, dx, dy: Double;
Wa, Wb, Clearance: Double;
EndPt: PointType;
begin
for i := 1 to Node[NodeNum].NumConnected do
begin
Clearance := WeldClearance * Member[Node[NodeNum].MemberEnd[i].member].CrossSection.t;

if (Member[Node[NodeNum].MemberEnd[i].Member].Node1 = NodeNum) then
begin
EndPt := Node[Member[Node[NodeNum].MemberEnd[i].Member].Node2].Location;
a := Member[Node[NodeNum].MemberEnd[i].member].CrossSection.y;
b := Member[Node[NodeNum].MemberEnd[i].member].CrossSection.l1 - a;
Wa := Member[Node[NodeNum].MemberEnd[i].member].Weld2;
Wb := Member[Node[NodeNum].MemberEnd[i].member].Weld3;
end
else
begin
EndPt := Node[Member[Node[NodeNum].MemberEnd[i].Member].Node1].Location;
b := Member[Node[NodeNum].MemberEnd[i].member].CrossSection.y;
a := Member[Node[NodeNum].MemberEnd[i].member].CrossSection.l1 - b;
Wa := Member[Node[NodeNum].MemberEnd[i].member].Weld3;
Wb := Member[Node[NodeNum].MemberEnd[i].member].Weld2;
end;

ang := angle(Node[NodeNum].Location, EndPt);
ang := ang * pi / 180;
dx := (Node[NodeNum].MemberEnd[i].CropDist - Clearance) * Cos(ang);
dy := (Node[NodeNum].MemberEnd[i].CropDist - Clearance) * Sin(ang);
a := a + Clearance;
b := b + Clearance;

Node[NodeNum].MemberEnd[i].Poly[1].x := Node[NodeNum].Location.x + dx + a * cos(ang - pi / 2);
Node[NodeNum].MemberEnd[i].Poly[1].y := Node[NodeNum].Location.y + dy + a * sin(ang - pi / 2);

Node[NodeNum].MemberEnd[i].Poly[2].x := Node[NodeNum].Location.x + dx - b * cos(ang - pi / 2);
Node[NodeNum].MemberEnd[i].Poly[2].y := Node[NodeNum].Location.y + dy - b * sin(ang - pi / 2);

Node[NodeNum].MemberEnd[i].Poly[3].x := Node[NodeNum].Location.x + dx - b * cos(ang - pi / 2) +
Wb * Cos(ang);
Node[NodeNum].MemberEnd[i].Poly[3].y := Node[NodeNum].Location.y + dy - b * sin(ang - pi / 2) +
Wb * Sin(ang);

Node[NodeNum].MemberEnd[i].Poly[4].x := Node[NodeNum].Location.x + dx + a * cos(ang - pi / 2) +
Wa * Cos(ang);

```

```
Node[NodeNum].MemberEnd[i].Poly[4].y := Node[NodeNum].Location.y + dy + a * sin(ang - pi / 2) +  
Wa * Sin(ang);
```

```
Node[NodeNum].MemberEnd[i].Poly[5] := Node[NodeNum].MemberEnd[i].Poly[1];  
end;  
end;
```

```
procedure BuildMemberPolygons;
```

```
{simply builds the member polygons for all of the members at once}
```

```
var
```

```
  i: integer;
```

```
  ang, a, b: Double;
```

```
begin
```

```
  for i := 1 to NumMembers do
```

```
    begin
```

```
      ang := angle(Node[Member[i].Node1].Location, Node[Member[i].Node2].Location);
```

```
      ang := ang * pi / 180;
```

```
      a := Member[i].CrossSection.y;
```

```
      b := Member[i].CrossSection.l1 - a;
```

```
      Member[i].Poly[1].x := Node[Member[i].Node1].Location.x + a * cos(ang - pi / 2);
```

```
      Member[i].Poly[1].y := Node[Member[i].Node1].Location.y + a * sin(ang - pi / 2);
```

```
      Member[i].Poly[2].x := Node[Member[i].Node1].Location.x - b * cos(ang - pi / 2);
```

```
      Member[i].Poly[2].y := Node[Member[i].Node1].Location.y - b * sin(ang - pi / 2);
```

```
      Member[i].Poly[3].x := Node[Member[i].Node2].Location.x - b * cos(ang - pi / 2);
```

```
      Member[i].Poly[3].y := Node[Member[i].Node2].Location.y - b * sin(ang - pi / 2);
```

```
      Member[i].Poly[4].x := Node[Member[i].Node2].Location.x + a * cos(ang - pi / 2);
```

```
      Member[i].Poly[4].y := Node[Member[i].Node2].Location.y + a * sin(ang - pi / 2);
```

```
      Member[i].Poly[5] := Member[i].Poly[1];
```

```
    end;
```

```
  end.
```

```
unit Optimize;
```

```
interface
```

```
uses
```

```
DataStructure, Utilities, SegmentIntersection, LayoutGeometry, GussetPlateGeometry;
```

```
function IntersectionPtSet (Poly1, Poly2: SubPolygonType): PolygonType;
```

```
function GussetPlateArea (X: XType): Double;
```

```
function ConstraintAreas (X: XType): GType;
```

```
{****}
```

```
{this unit contains the procedures that would be used by an optimizer to optimize the gusset}
```

```
{plate area. These are older versions of ones that Mehran Chirehdast and I have modified}
```

```
{an changed to use on the appollos. In this program, the intersectionptset function is called}
```

```
{to do the cropping}
```

```
implementation
```

```
function IntersectionPtSet (Poly1, Poly2: SubPolygonType): PolygonType;
```

```
{given two subpolygons, it finds the pts that belong to the intersection polygopmn}
```

```
{of the two. These pts must then be ordered by orderptset to get the polygon.}
```

```
var
```

```
  i, j, Size: integer;
```

```
  intercase: IntersectionType;
```

```
  InterPt, pt1, pt2, pt3, pt4: PointType;
```

```
  PtSet: PolygonType;
```

```
begin
```

```
  Size := 0;
```

```
  for i := 1 to 4 do
```

```
    if PtInsidePolygon(Poly1[i], Poly2) or PtOnPolygon(Poly1[i], Poly2) then
```

```
      begin
```

```
        Size := Size + 1;
```

```
        PtSet.Pts[Size] := Poly1[i];
```

```
      end;
```

```
  for i := 1 to 4 do
```

```
    if PtInsidePolygon(Poly2[i], Poly1) or PtOnPolygon(Poly2[i], Poly1) then
```

```
      begin
```

```
        Size := Size + 1;
```

```
        PtSet.Pts[Size] := Poly2[i];
```

```
      end;
```

```
  for i := 1 to 4 do
```

```
    begin
```

```
      pt1 := Poly1[i];
```

```
      pt2 := Poly1[i + 1];
```

```
      for j := 1 to 4 do
```

```
        begin
```

```
          pt3 := Poly2[j];
```

```
          pt4 := Poly2[j + 1];
```

```
          SegmentIntersection(pt1, pt2, pt3, pt4, intercase, InterPt);
```

```
          if Intercase = MidPoint then
```

```
            begin
```

```
              Size := Size + 1;
```

```
              PtSet.Pts[Size] := InterPt;
```

```

    end;
    end;
    end;

PtSet.Size := Size;
IntersectionPtSet := PtSet;
end;

function ConstraintAreas (X: XType): GType;
var
    i, j, k, Mnum1, Mnum2: integer;
    G: Gtype;
    TempArea: Double;
begin
    for i := 1 to Node[CurrentNode].NumConnected do
        begin
            Node[CurrentNode].MemberEnd[i].CropDist := X[i];
            CropMemberEnd(Node[CurrentNode].MemberEnd[i].Member, CurrentNode,
Node[CurrentNode].MemberEnd[i].CropDist);
            end;

        k := 0;
        for i := 1 to Node[CurrentNode].NumConnected do
            for j := i to Node[CurrentNode].NumConnected do
                begin
                    k := k + 1;
                    Mnum1 := Node[CurrentNode].MemberEnd[i].Member;
                    Mnum2 := Node[CurrentNode].MemberEnd[j].Member;
                    TempArea := PolygonArea(OrderPtSet(IntersectionPtSet(Member[Mnum1].poly,
Member[Mnum2].Poly)));
                    G[k] := TempArea * TempArea;
                    end;
                ConstraintAreas := G;
            end;
        end;

function GussetPlateArea (X: XType): Double;
var
    i: integer;
begin
    for i := 1 to Node[CurrentNode].NumConnected do
        begin
            Node[CurrentNode].MemberEnd[i].CropDist := X[i];
            CropMemberEnd(Node[CurrentNode].MemberEnd[i].Member, CurrentNode,
Node[CurrentNode].MemberEnd[i].CropDist);
            end;

        Node[CurrentNode].GussetPlate := RemoveConcavePts(OrderPtSet(FindGussetPlatePts(CurrentNode)));
        GussetPlateArea := PolygonArea(Node[CurrentNode].GussetPlate);
        end;
    end.

```

```

unit Examples;

interface
uses
  DataStructure;

procedure HardCodeThree;
procedure HardCodeFour;
procedure HardCodeSupport;

{****}
{this unit contains some hardcoded examples. all of the numbers come from the one example}
{in the chapter written by ben, julie, and mehi with Proff. Papalambros so that the program}
{could be checked to see if it was working}

{all of the units are [mm] and [kN] :-)      }
{****}
implementation

procedure HardCodeThree;
begin
  NumNodes := 3;
  NumMembers := 3;

  with Node[1] do
  begin
    location.x := 1000;
    location.y := 1500;
    NumConnected := 2;
    MemberEnd[1].Member := 1;      {this order determines the order cropping happens in}
    MemberEnd[2].Member := 3;      {if the order were 3, 1 instead of 1,3 it would happen}
  end;                               {in reverse}

  with Node[2] do
  begin
    location.x := 1000;
    location.y := 3000;
    NumConnected := 2;
    MemberEnd[1].Member := 1;
    MemberEnd[2].Member := 2;
  end;

  with Node[3] do
  begin
    location.x := 3000;
    location.y := 3000;
    NumConnected := 2;
    MemberEnd[1].Member := 2;
    MemberEnd[2].Member := 3;
  end;

  with Member[1] do
  begin
    Node1 := 1;
    Node2 := 2;

```

```
Load := 222.4;  
CrossSection.L1 := 127;  
CrossSection.L2 := 76;  
CrossSection.y := 43;  
CrossSection.t := 9.5;  
end;
```

```
with Member[2] do  
begin  
Node1 := 2;  
Node2 := 3;  
Load := 222.4;  
CrossSection.L1 := 127;  
CrossSection.L2 := 76;  
CrossSection.y := 43;  
CrossSection.t := 9.5;  
end;
```

```
with Member[3] do  
begin  
Node1 := 1;  
Node2 := 3;  
Load := 222.4;  
CrossSection.L1 := 127;  
CrossSection.L2 := 76;  
CrossSection.y := 43;  
CrossSection.t := 9.5;  
end;  
end;
```

```
procedure HardCodeFour;  
begin  
NumNodes := 4;  
NumMembers := 5;
```

```
with Node[1] do  
begin  
location.x := 0;  
location.y := 0;  
NumConnected := 2;  
MemberEnd[1].Member := 1;  
MemberEnd[2].Member := 4;  
end;
```

```
with Node[2] do  
begin  
location.x := 0;  
location.y := 1500;  
NumConnected := 3;  
MemberEnd[1].Member := 5;  
MemberEnd[2].Member := 2;  
MemberEnd[3].Member := 1;  
end;
```

```
with Node[3] do  
begin
```

```
location.x := 3000;
location.y := 1500;
NumConnected := 2;
MemberEnd[1].Member := 2;
MemberEnd[2].Member := 3;
end;
```

```
with Node[4] do
begin
location.x := 3000;
location.y := 0;
NumConnected := 3;
MemberEnd[1].Member := 5;
MemberEnd[2].Member := 3;
MemberEnd[3].Member := 4;
end;
```

```
with Member[1] do
begin
Node1 := 1;
Node2 := 2;
Load := 222.4;
CrossSection.L1 := 127;
CrossSection.L2 := 76;
CrossSection.y := 43;
CrossSection.t := 9.5;
end;
```

```
with Member[2] do
begin
Node1 := 2;
Node2 := 3;
Load := 222.4;
CrossSection.L1 := 127;
CrossSection.L2 := 76;
CrossSection.y := 43;
CrossSection.t := 9.5;
end;
```

```
with Member[3] do
begin
Node1 := 3;
Node2 := 4;
Load := 222.4;
CrossSection.L1 := 127;
CrossSection.L2 := 76;
CrossSection.y := 84;
CrossSection.t := 9.5;
end;
```

```
with Member[4] do
begin
Node1 := 1;
Node2 := 4;
Load := 222.4;
CrossSection.L1 := 127;
```

```
CrossSection.L2 := 76;  
CrossSection.y := 43;  
CrossSection.t := 9.5;  
end;
```

```
with Member[5] do  
begin  
Node1 := 2;  
Node2 := 4;  
Load := 222.4;  
CrossSection.L1 := 127;  
CrossSection.L2 := 76;  
CrossSection.y := 84;  
CrossSection.t := 9.5;  
end;  
end;
```

```
procedure HardCodeSupport;  
begin  
NumNodes := 6;  
NumMembers := 8;
```

```
with Node[1] do  
begin  
location.y := 900;  
location.x := 4300;  
NumConnected := 3;  
MemberEnd[1].Member := 3; {this order determines the order of cropping}  
MemberEnd[2].Member := 2;  
MemberEnd[3].Member := 1;  
end;
```

```
with Node[2] do  
begin  
location.y := 2100;  
location.x := 3500;  
NumConnected := 4;  
MemberEnd[1].Member := 6;  
MemberEnd[2].Member := 5;  
MemberEnd[3].Member := 4;  
MemberEnd[4].Member := 3;  
end;
```

```
with Node[3] do  
begin  
location.y := 3300;  
location.x := 4300;  
NumConnected := 3;  
MemberEnd[1].Member := 4;  
MemberEnd[2].Member := 7;  
MemberEnd[3].Member := 8;  
end;
```

```
with Node[4] do
```

```
begin
location.y := 0;
location.x := 0;
NumConnected := 2;
MemberEnd[1].Member := 5;
MemberEnd[2].Member := 1;
end;
```

```
with Node[5] do
begin
location.y := 4000;
location.x := 0;
NumConnected := 2;
MemberEnd[1].Member := 6;
MemberEnd[2].Member := 7;
end;
```

```
with Node[6] do
begin
location.y := 2100;
location.x := 6400;
NumConnected := 2;
MemberEnd[1].Member := 2;
MemberEnd[2].Member := 8;
end;
```

```
with Member[1] do
begin
Node1 := 1;
Node2 := 4;
Load := 222.4;
CrossSection.L1 := 127;
CrossSection.L2 := 76;
CrossSection.y := 84;
CrossSection.t := 9.5;
end;
```

```
with Member[2] do
begin
Node1 := 1;
Node2 := 6;
Load := 222.4;
CrossSection.L1 := 127;
CrossSection.L2 := 76;
CrossSection.y := 43;
CrossSection.t := 9.5;
end;
```

```
with Member[3] do
begin
Node1 := 1;
Node2 := 2;
Load := 222.4;
CrossSection.L1 := 127;
CrossSection.L2 := 76;
```

```
CrossSection.y := 43;
CrossSection.t := 9.5;
end;

with Member[4] do
begin
Node1 := 2;
Node2 := 3;
Load := 222.4;
CrossSection.L1 := 127;
CrossSection.L2 := 76;
CrossSection.y := 43;
CrossSection.t := 9.5;
end;

with Member[5] do
begin
Node1 := 2;
Node2 := 4;
Load := 222.4;
CrossSection.L1 := 127;
CrossSection.L2 := 76;
CrossSection.y := 84;
CrossSection.t := 9.5;
end;

with Member[6] do
begin
Node1 := 2;
Node2 := 5;
Load := 222.4;
CrossSection.L1 := 127;
CrossSection.L2 := 76;
CrossSection.y := 43;
CrossSection.t := 9.5;
end;

with Member[7] do
begin
Node1 := 3;
Node2 := 5;
Load := 222.4;
CrossSection.L1 := 127;
CrossSection.L2 := 76;
CrossSection.y := 43;
CrossSection.t := 9.5;
end;

with Member[8] do
begin
Node1 := 3;
Node2 := 6;
Load := 222.4;
CrossSection.L1 := 127;
CrossSection.L2 := 76;
```

```
CrossSection.y := 84;  
CrossSection.t := 9.5;  
end;  
end;  
end.
```

