G

Graphic Utilities

TABLE OF CONTENTS

§G.1. Introduction

This Appendix collects a set of application-independent graphic utilities organized as *Mathematica* modules. By "application independent" is meant that these modules are reusable across different FEM application programs that do not necessarily simulate structures.

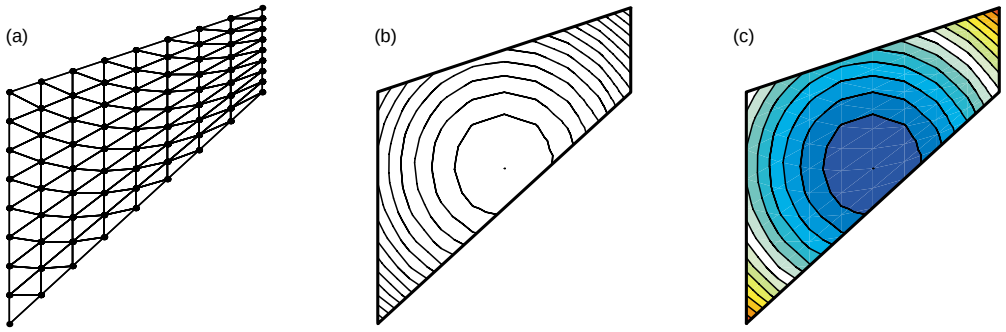

FIGURE G.1. Two-dimensional FEM plot types: (a) mesh plot; (b) contour line plot (CLP); (c) contour band plot (CBP); (d) contour polygonal plot (CPP), to be eventually added.

§G.1.1. Finite Element Plot Types

Figure G.1 illustrates several types of graphic displays pertaining to a two-dimensional (2D) finite element model. The mesh plot in Figure G.1(a) shows elements and nodes; in this case a mesh of 3-node triangles. Contour plots, such as those shown in Figure G.1(b)-(d) are a commonly used graphical technique to display a function of two variables defined over a 2D mesh. Distinguishing among these types requires some terminology, which is introduced in the next subsection.

§G.1.2. Contour Plotting Terminology

A *contour line*, also called *isoline*, of a function of two variables is a curve along which the function has a constant value. In cartography, a contour line, sometimes just called a *contour*, joins points of equal elevation (height) above a reference level, such as mean sea level. A *contour map*, also called *topographical map*, is a map illustrated with contour lines. The *contour interval* of a contour map is the difference in values between successive contour lines. The generalization of a contour line to functions of more than two variables is called a *level set*.

Now we are in a position to describe the three contour plot types shown in Figure G.1.

1. *Contour line plot*, or CLP. As shown in Figure G.1(b) this is just that: a set of contour lines, typically drawn at equal contour intervals. No color appears between lines. This kind of display is primarily used to identify general features at a glance; for example, steep gradient regions.
2. *Contour band plot*, or CBP. This is produced by coloring the contour bands (regions comprised between succesive contour line) as pictured in Figure G.1(c). The color is picked as per a one-parameter value-to-color mapping scheme, in which "value" means the averaged value of the function at the band. This form is more frequently used than CLP, since it conveys more information at a glance. In particular, regions of maximum and minimum values are easily identified by color.

3. *Contour polygonal plot*, or CPP. Finite element domains are subdivided into polygons, each each of which is filled with a color again defined by a value-to-color mapping, in which "value" means the average function value over the polygon. This type is better suited to display of discontinuous functions that may jump between elements, but it has the disadvantage of rapid (superlinear) growth in cost as the number of subdivisions over each element increases.

Both CLP and CBP are not restricted to finite element meshes but can be used for other applications. For example CLP is often used in cartography to plot elevations over a reference height. On the other hand, CPP is (by nature) restricted to FEM since it relies on subdividing element regions.

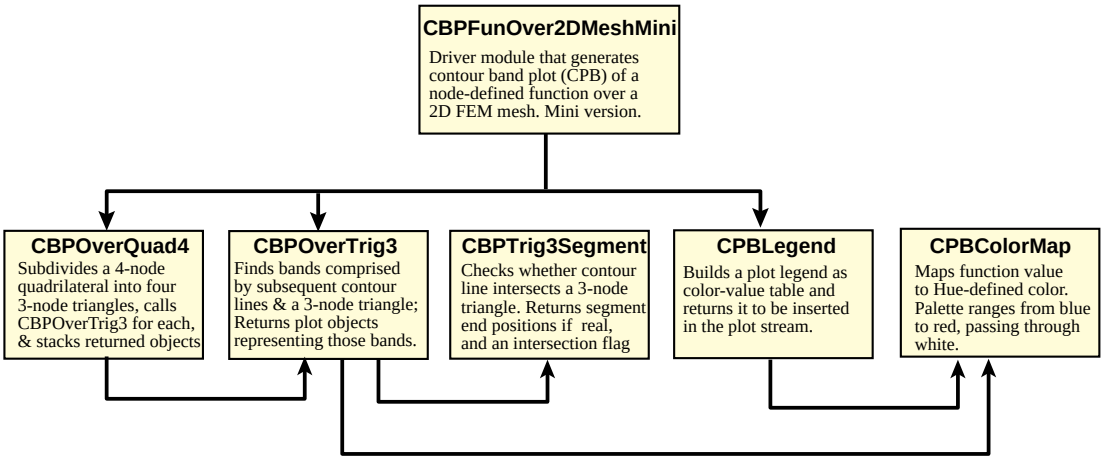

FIGURE G.2. Configuration of CBP plotter software with `CBPOver2DMeshMini` as driver.

§G.2. Contour Band Plotter: Mini Version

This section describes a "mini" CBP version with very basic contour plotting capabilities. Specific restrictions are:

1. Only 3-node triangles and 4-node quadrilaterals are accepted
2. Contour line interval is fixed
3. The plot object cannot be returned for downstream processing
4. The set of graphic options is limited.

The plotter software comprises six *Mathematica* modules, which are configured as shown in the diagram of Figure G.3. A full version is described in another Section. The full version allows more options, and accepts a richer set of element configurations. Both versions allow display of legends.

Both versions: mini and full, have an important limitation:

Function to be plotted is assumed C^0 continuous over the finite element mesh.

The main reason behind this restriction is that function values are given at nodes, and the value is assumed to be the same among all 2D elements connected to that node. If the function exhibits interelement jump discontinuities (for example, at material interfaces) a contour polygonal plotter (CPP) that allows functions to be specified element-by-element is more appropriate. One such module is described in another Section. (Eventually.)

```
CBPOver2DMeshMini[nodxyz_,elenod_,eletyp_,fn_,frain_,plotwhat_,
  plotspecs_]:=Module[{e,i,n,nc,xyc,fc,enl,c,pb,pl,fmin,fmax,ncint,feps,
  fspecs,colors,mesh,nodes,bacg,lines,legspecs,aspect,label,asprat,
  prebacg,pbacg,preband,pband,preline,pline,premesh,pmesh,prenode,pnode,
  preleg,pleg,empty,numele=Length[elenod]}, empty=Table[{},{numele}];
  {colors,mesh,nodes,bacg,lines}=plotwhat;
  pband=pmesh=pnode=pline=pbacg=preleg=pleg={};
  {legspecs,aspect,label}=plotspecs; {fmin,fmax,ncint}=frain;
  feps=10.^(-8)*Max[Abs[fmax],Abs[fmin]]; fspecs={fmin,fmax,ncint,feps};
  If [colors,pband=empty]; If [mesh, pmesh=empty];  If [bacg,pbacg=empty];
  If [lines, pline=empty]; If [nodes,pnode=empty];
  prebacg={Graphics[RGBColor[.7,.7,.7]]}; preband={};
  preline={Graphics[RGBColor[0,0,0]],Graphics[AbsoluteThickness[1]]};
  premesh={Graphics[RGBColor[1,0,0]],Graphics[AbsoluteThickness[2]],
           Graphics[AbsoluteDashing[{1,10}]]};
  prenode={Graphics[RGBColor[0,0,0]],Graphics[AbsoluteDashing[{}]],
           Graphics[AbsolutePointSize[4]]};
  For [e=1,e<=numele,e++,
      enl=elenod[[e]]; nc=Length[enl];
      If [!MemberQ[{3,4},nc], Print["CBP mini:",
           "Elem ",e," with ",nc," nodes skipped"]; Continue[]];
      fc=xyc=Table[0,{nc}];
      Do [n=enl[[i]]; xyc[[i]]=nodxyz[[n]];fc[[i]]=fn[[n]],{i,1,nc}];
      If [nc==3, {pb,pl}=CBPOverTrig3[xyc,fc,fspecs,lines];
          c=Table[xyc[[{1,2,3,1}[[i]]]],{i,4}]];
      If [nc==4, {pb,pl}=CBPOverQuad4[xyc,fc,fspecs,lines];
          c=Table[xyc[[{1,2,3,4,1}[[i]]]],{i,5}]];
      If [colors, pband[[e]]=pb]; If [lines, pline[[e]]=pl];
      If [mesh,   pmesh[[e]]=Graphics[Line[c]]];
      If [bacg,   pbacg[[e]]=Graphics[Polygon[c]]];
      If [nodes,  pnode[[e]]=Table[Graphics[Point[ xyc[[i]] ]],
          {i,nc}]];
      ];
 asprat=aspect; If [aspect=={}, asprat=Automatic];
 If [!colors,pband={}]; If [!mesh, pmesh={}]; If [!nodes, pnode={}];
 If [!lines, pline={}]; If [!bacg, pbacg={}]; pleg={};
 If [Length[legspecs]==0||!colors, Show[prebacg,pbacg,
     preband,pband,preline,pline,premesh,pmesh,prenode,
     pnode,AspectRatio->asprat,PlotLabel->label,
     DisplayFunction->DisplayChannel ]];
 If [Length[legspecs]>0&&colors,
     {preleg,pleg}=CBPLegend[{fmin,fmax},legspecs,1];
     Show[prebacg,pbacg,preband,pband,preline,pline,
          premesh,pmesh,prenode,pnode,preleg,pleg,
          TextStyle->{FontFamily->"Courier",
                  FontSlant->"Plain", FontSize->10},
          AspectRatio->asprat,PlotLabel->label,
          DisplayFunction->DisplayChannel ]];
 ClearAll[pband,pline,pmesh,pnode,pbacg,pleg]  ];
```

FIGURE G.3. Module `CBPOver2DMeshMini` to produce a contour band plot (CBP) of a function over a 2D finite element version. Mini version.

§G.2.1. Plot Driver: CBPOver2DMeshMini

The driver program is called `CBPOver2DMeshMini`. The code is listed in Figure G.3. It is invoked by

```
CBPOver2DMeshMini[nodxyz,elenod,eletyp,fn,frain,plotwhat,plotspecs]   (G.1)
```

The arguments are:

nodxyz — {x, y} Node coordinates stored as {{x1,y1},{x2,y2}, ...,{xN,yN}}. in which N is number of nodes. The z coordinate is assumed zero and need not be listed,

elenod — Element node lists, stored as {enl1}, {enl2}, ...,{enlNe}}, in which Ne is number of elements. For a 3-node triangle, enl={n1,n2,n3}. For a 4-node quadrilateral, enl={n1,n2,n3,n4}. Element node lists that do not contain 3 or 4 nodes are skipped, and a warning message given.

eletyp — A dummy argument. In the full CBP version, it contains element type identifiers.[1]

fn — Function values at nodes: {f1,f2, ... fN}. The value is assumed to be the same for all elements connected to node.

frain — A 3-entry list {fmin,fmax,ncint} specifying function range and contour interval:

fmin,fmax: minimum and maximum function values to appear in plot

ncint: number of contour intervals in the range fmin through fmax.

The contour interval will be finc=(fmax-fmin)/ncint. If fmax $\leq$ fmin or ncint$\leq$0, contour lines and bands are not plotted. Regions where the function value is outside the range will appear black.

plotwhat — A list of five logical flags: {colors, mesh, nodes, bacg, lines}. These flags specify what is to be plotted:

colors: If True, show band colors; if False omit.

mesh: If True, show background mesh in dashed lines; if False omit.

nodes: If True, show mesh nodes as black dots, if False omit.

bacg: If True, show boundaries; if False omit.

lines: If True, show contour lines that separate bands; if False omit.

Note: this argument is more elaborate in the full CBP version.

plotspecs — A set of specifications: {legspec,aspect,label} that controls appearance:

legspec: Plot legend specifications: {{xleg,yleg},mv}, in which {xleg,yleg} are coordinates of a location where the legend frame is to appear, and mv is number of value intervals to shown in legend. To omit legend specify an empty list: { }. For legend placement and configuration details see §G.2.5.

aspect: Aspect ratio of plot frame. If zero or negative, the default setting AspectRatio->Automatic is used.

label: A textstring to be placed as plot label. To omit specify an empty list: { }.

The module does not return a value. To skip the plot display and have the CBP object returned for downstream processing, the full version should be used.

The major plotting operations performed by the driver module are carried out inside a For loop over the elements (see Figure G.3 for coding details; loop runs on element index e). The only elements accepted by the mini version are 3-node triangles and 4-node quadrilaterals. The latter are split into four 3-node triangles forming a Union Jack pattern, as described in §G.2.4. So eventually it

[1] The full version has exactly the same arguments, but some of them are more complicated.

```
CBPOverTrig3[xyc_,fc_,fspecs_,lines_]:=Module[{fmin,fmax,ncint,
  feps,nv,iv,xc1,yc1,xc2,yc2,xc3,yc3,fc1,fc2,fc3,xs,ys,fs,
  x1,x2,x3,y1,y2,y3,f1,f2,f3,firsthit,finc,Pc1,Pc2,Pc3,
  ftop,fbot,favg,Pb,Pt,tb,tt,P1,P2,P3,P4,Q1,Q2,Q3,Q4,ptab,
  flast,frange,Plast,tlast,kb,kl,p,pb,pl},
 {fmin,fmax,ncint,feps}=fspecs; nv=Min[ncint,1000];
  If [fmax<fmin||ncint<=0, Return[{{},{}}]];
  pb=Table[{},{nv}]; pl=Table[{},{nv+1}]; frange={fmin,fmax};
 {{xc1,yc1},{xc2,yc2},{xc3,yc3}}=N[xyc]; {fc1,fc2,fc3}=N[fc];
 {{x1,y1,f1},{x2,y2,f2},{x3,y3,f3}}=Sort[{{xc1,yc1,fc1},
       {xc2,yc2,fc2},{xc3,yc3,fc3}},#1[[3]]<=#2[[3]]&];
  xs={x1,x2,x3}; ys={y1,y2,y3}; fs={f1,f2,f3};
  Pc1={x1,y1}; Pc2={x2,y2}; Pc3={x3,y3};
  ptab={{{},{Pc1,Pc2,Pc3},{Pc1,Q4,Q3},{Pc1,Pc2,Q3,Q4}},
     {{Null},{},{Null},{Null}},
     {{Null},{Pc3,Pc2,Q1,Q2},{Q1,Q2,Q4,Q3},{Pc2,Q1,Q2,Q4,Q3}},
     {{Null},{Pc3,Q1,Q2},{Null},{Q1,Q2,Q4,Q3}}};
  finc=(fmax-fmin)/nv; flast=fmin-1000; firsthit=True; kb=kl=0;
  For [iv=1,iv<=nv,iv++,
       fbot=N[fmin+(iv-1)*finc]; ftop=N[fbot+finc];
       If [ftop<=f1 || fbot>=f3, Continue[]]; favg=(fbot+ftop)/2;
       If [fbot==flast, {fbot,Pb,tb}={flast,Plast,tlast},
          {Pb,tb}=CBPTrig3Segment[xs,ys,fs,fbot,feps]];
       {Pt,tt}=CBPTrig3Segment[xs,ys,fs,ftop,feps];
       {P1,P2}=Pb; {P3,P4}=Pt; {flast,Plast,tlast}={ftop,Pt,tt};
       If [lines,
          If [firsthit&&(tb>0),
              pl[[++kl]]=Graphics[Line[Pb]]; firsthit=False];
          If [tt>0, pl[[++kl]]=Graphics[Line[Pt]]]
          ];
       p=ptab[[tb+2,tt+2]]/.{Q1->P1,Q2->P2,Q3->P3,Q4->P4};
       pb[[++kb]]={CBPColorMap[favg,frange,1],Graphics[Polygon[p]]};
       ];
  If [kb>0,pb=Take[pb,kb],{},{}]; If [kl>0,pl=Take[pl,kl],{},{}];
  ClearAll[ptab]; Return[{pb,pl}] ];
```

FIGURE G.4. Module `CBPOverTrig3` that contour-bands a function over a 3-node triangle.

all boils down to contour-banding over 3-node triangles; this is done as described in §G.2.2. The reason for this splitting is that the function $f(x, y)$ can be then linearly interpolated from the 3 corner values $\{f_1, f_2, f_3\}$.

§G.2.2. Band-Contour 3-Node Triangle: CBPOverTrig3

Module CBPOverTrig3 searches over the plot range for all contour lines that cross a specific 3-node triangle, and builds the corresponding contour band polygons.[2] The module source code is listed in Figure G.4. The module is invoked as

```
{preleg,plab}= CBPOverTrig3[xyc,fc,fspecs,lines]
```
(G.2)

The arguments are:

xyc	The $\{x, y\}$ corner coordinates arranged `{{x1,y1},{x2,y2},{x3,y3}}`.
fc	Corner function values stored as `{f1,f2,f3}`. Corners are internally renumbered within the module so that `f1`≤`f2`≤`f3`.
fspecs	A 4-item list of function plotting specifications: `{fmin,fmax,ncint,feps}`.

[2] This operation is called *polygon clipping* in the computer science literature.

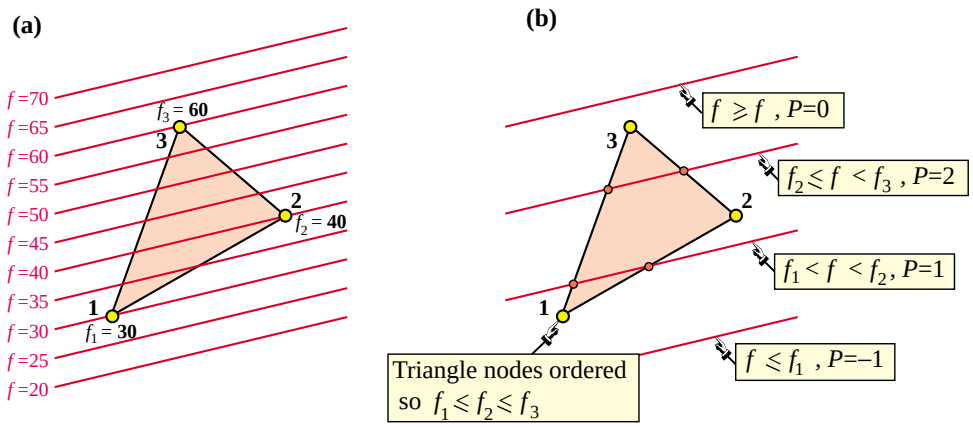

FIGURE G.5. Contour lines of a plane passing through triangle corners with values (sorted by increasing value) are `f1=30`, texttf2=40 and `f3=60`. (a) Contour lines `f=20` through `f=70` are straight lines on account of linear interpolation from corner values (lines shown outside triangle for visualization convenience); (a) 4 intersection cases identified by an integer flag `P`.

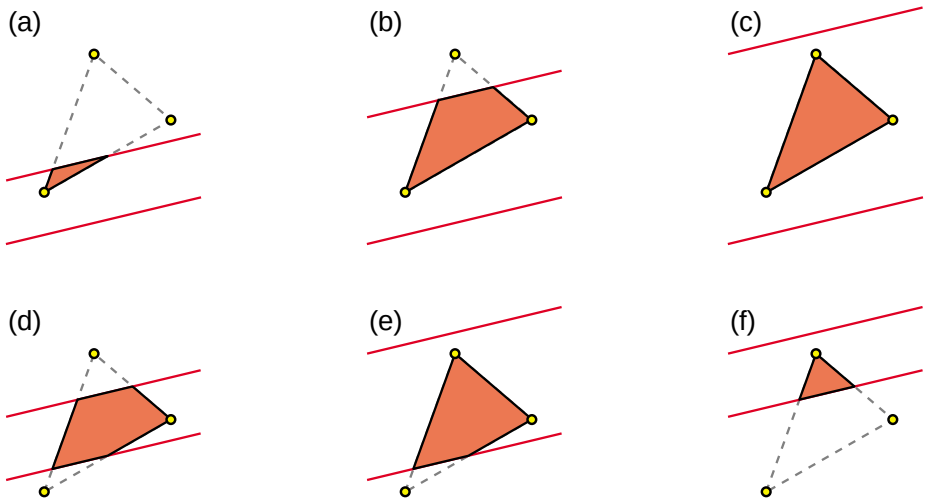

FIGURE G.6. Contour band polygons over a 3-node triangle. Shown are 6 cases in which one or more contour line intersections occur. Polygons have 3, 4, 3, 5, 4 and 3 sides for cases (a) through (f), respectively.

`fmin,fmax,ncint`: Same as items in argument `frain` of the driver module.

`feps`: A function value tolerance set by the driver module `CBPOver2DMeshMini` to `eps*Max[Abs[fmin],Abs[fmax]]`, in which `eps<<1` is typically 10^{-8}. Used to avoid ill conditioned computations in `CBPTrig3Segment`.

lines — The logical flag supplied in argument `plotwhat` of driver module `CBPOver2DMeshMini`. If `True`, contour lines separating bands are drawn.

The module returns

`{pb,pl}` — Graphic objects that define contour band polygons if any found. The first object: `pb` has information about the polygon geometry and color fill, whereas the second object: textttpl, return polygon boundary lines if requested. If no contour band polygons are found, the objects are empty.

```
CBPTrig3Segment[xs_,ys_,fs_,fv_,feps_]:=Module[{x1,x2,x3,
  y1,y2,y3,f1,f2,f3,x21,y21,x32,y32,x13,y13,f21,f32,f31,
  ζ211,ζ212,ζ311,ζ313,ζ322,ζ323,P1,P3,P21,P32,P13},
  {x1,x2,x3}=xs; {y1,y2,y3}=ys; {f1,f2,f3}=fs; P1={x1,y1}; P3={x3,y3};
  If [fv<f1, Return[{{P1,P1},-1}]]; If [fv>f3, Return[{{P3,P3},0}]];
  f21=f2-f1; If [f21<feps,f1=f1-feps;f21=feps];
  f32=f3-f2; If [f32<feps,f3=f2+feps;f32=feps]; f31=f3-f1;
  ζ212=(fv-f1)/f21; ζ212=Max[Min[ζ212,1],0]; ζ211=1-ζ212;
  ζ313=(fv-f1)/f31; ζ313=Max[Min[ζ313,1],0]; ζ311=1-ζ313;
  P21={x2*ζ212+x1*ζ211,y2*ζ212+y1*ζ211};
  P13={x1*ζ311+x3*ζ313,y1*ζ311+y3*ζ313};
  If [fv<=f2,Return[{{P21,P13},1}]];
  ζ323=(fv-f2)/f32; ζ323=Max[Min[ζ323,1],0]; ζ322=1-ζ323;
  P32={x3*ζ323+x2*ζ322,y3*ζ323+y2*ζ322};
  Return[{{P32,P13},2}]];
```

FIGURE G.7. Module `CBPTrig3Segment` that determines whether a contour line intersects a 3-node triangle, and returns intersection segment information if that is the case.

The contour-band capturing logic[3] can be better explained through examination of Figures G.5 and G.6. Module `CBPOverTrig3` cycles over the set of contour line values and calls module `CBPTrig3Segment` (described in the next Subsection) to determine whether one or more intersections occur. If this the case, it uses a table-driven scheme to construct the appropriate polygon from the six cases shown in G.6.

§G.2.3. Contour-Line-Segment 3-Node Triangle: CBPTrig3Segment

Module CBPTrig3Segment is invoked by `CBPOverTrig3`. It receives the corner coordinates of a specific 3-node triangle, the corner function values sorted in ascending order, and a contour line value. It finds whether that line intersects the triangle, and returns appropriate information to that effect. The module source code is listed in Figure G.7. The module is invoked as

```
{xint,flag}= CBPTrig3Segment[xs,ys,fs,fv,feps]
```
(G.3)

The arguments are:

xs	The sorted x corner coordinates {`x1,x2,x3`}.
ys	The sorted y corner coordinates {`y1,y2,y3`}.
fs	The sorted corner function values {`f1,f2,f3`}.
fv	The countour line value.
feps	A function tolerance value used to avoid ill-conditioned computations. See `feps` argument in `CBPOverTrig3`.

The module returns

{{`P1,P2`},`P`}	If an intersection found, `P1,P2` contain the $\{x, y\}$ coordinates of the end points of the segment. If no intersection is found, both `P1` and `P2` are set to the coordinates of the corner closest to the contour line. Item `P` is an "intersection tag": an integer with values -1 through 2 that identifies the 4 cases pictured in Figure G.5(b).

The `CBPTrig3Segment` logic relies heavily on the use of the triangular coordinates $\{\zeta_1, \zeta_2, \zeta_3\}$ introduced in Chapter 15. This is done to help numerical accuracy and stability because such coordinates are dimensionless and well scaled, as they valued in the range $[0, 1]$. On the other hand, the Cartesian coordinates might be poorly scaled in some unit systems.

[3] The table-driven logic in `CBPOverTrig3` is admittedly hairy but can be understood after some study.

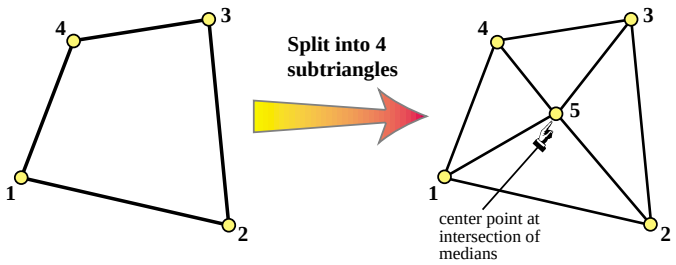

FIGURE G.8. Splitting a 4-node quadrilateral into four 3-node triangles.

```
CBPOverQuad4[xyc_,fc_,fspecs_,lines_]:=Module[
  {x1,x2,x3,x4,x5,y1,y2,y3,y4,y5,f1,f2,f3,f4,f5,
   e,xytab,ftab,xyt,ft, pbq,plq,pbt,plt},
   pbq=plq=Table[{},{4}];
   {{x1,y1},{x2,y2},{x3,y3},{x4,y4}}=xyc; {f1,f2,f3,f4}=fc;
   x5=(x1+x2+x3+x4)/4; y5=(y1+y2+y3+y4)/4; f5=(f1+f2+f3+f4)/4;
   xytab={{{x1,y1},{x2,y2},{x5,y5}},{{x2,y2},{x3,y3},{x5,y5}},
          {{x3,y3},{x4,y4},{x5,y5}},{{x4,y4},{x1,y1},{x5,y5}}};
   ftab={{f1,f2,f5},{f2,f3,f5},{f3,f4,f5},{f4,f1,f5}};
   For [e=1,e<=4,e++, xyt=xytab[[e]]; ft=ftab[[e]];
       {pbq[[e]],plq[[e]]}=CBPOverTrig3[xyt,ft,fspecs,lines];
       ];
   Return[{pbq,plq}] ];
```

FIGURE G.9. Module `CBPOverQuad4` that contour-bands a function over a 4-node quadrilateral.

§G.2.4. Band-Contour 4-Node Quadrilateral: CBPOverQuad4

Module `CBPOverQuad4` processes a 4-node quadrilateral element (Quad4) by subdividing it into four subtriangles that meet at the element center, as illustrated in Figure G.8. The average of the corner function values is assigned at the center node. It then calls `CBPTrig3` four times, and stacks the plot object information. The module source code is listed in Figure G.9.

The module is invoked as

```
{plq,pbq}= CBPOverQuad4[xyc,fc,fspecs,lines]
```
(G.4)

The arguments are:

xyc	The $\{x, y\}$ corner coordinates arranged `{{x1,y1},{x2,y2},{x3,y3},{x4,y4}}`.
fc	Corner function values stored as `{f1,f2,f3,f4}`.
fspecs	Same as in `CBPOverTrig3`.
lines	Same as in `CBPOverTrig3`.

The module returns

`{pbq,plq}`	Graphic objects that define contour band polygons if any found. The first object: `pbq` has information about the polygon geometry and color fill, whereas the second object: textttplq, returns polygon boundary lines if requested. If no contour band polygons are found, the objects are empty.

```
CBPLegend[frange_,legspecs_,smax_]:=Module[
  {fmin,fmax,f,fs,finc,fa,fb,xylinL,xylinR,xytext,
   black=Graphics[RGBColor[0,0,0]],preleg,plab={},
   white=Graphics[RGBColor[1,1,1]],xy1,xy2,xy3,xy4,
   baselineskip=10,dy,xf1,xf2,xf3,xf4,iv,xyleg,nvleg,
   thin= Graphics[AbsoluteThickness[1]],color,padOK},
   {xyleg,nvleg}=legspecs; nvleg=Min[Max[nvleg,1],20];
   {fmin,fmax}=frange; {fa,fb}={Abs[fmin],Abs[fmax]};
   xf1=Offset[{-30,  10},xyleg]; xf2=Offset[{85,10},xyleg];
   xf3=Offset[{ 85, -10-baselineskip*(nvleg+1)},xyleg];
   xf4=Offset[{-30, -10-baselineskip*(nvleg+1)},xyleg];
   preleg={thin,white,
       Graphics[Polygon[{xf1,xf2,xf3,xf4}]],black,
       Graphics[Line[{xf1,xf2,xf3,xf4,xf1}]],
       Graphics[Text["LEGEND",Offset[{5,0},xyleg],{-1,0}]]};
   plab=Table[{},{nvleg+1}]; If [nvleg<=0,Return[{preleg,plab}]];
   finc=(fmax-fmin)/nvleg; dy=baselineskip/2;
   padOK=(Max[fa,fb]<=1000)&&(Min[fa,fb]>=1/100);
   For [iv=1,iv<=nvleg+1,iv++, f=N[fmin+(iv-1)*finc];
        xy1=Offset[{-22,-baselineskip*iv+dy},xyleg];
        xy2=Offset[{-22,-baselineskip*iv-dy},xyleg];
        xy3=Offset[{  8,-baselineskip*iv-dy},xyleg];
        xy4=Offset[{  8,-baselineskip*iv+dy},xyleg];
        xytext=Offset[{ 80,-baselineskip*iv},xyleg];
        If [padOK, fs=PaddedForm[f,4,NumberSigns->{"-","+"}],
              fs=ScientificForm[f,4,NumberSigns->{"-","+"}]];
        color=CBPColorMap[f,{fmin,fmax},smax];
        plab[[iv]]={color,Graphics[Polygon[{xy1,xy2,xy3,xy4}]],
                    black,Graphics[Text[fs,xytext,{1,0}]]}];
   Return[{preleg,plab}]];
```

FIGURE G.10. Module `CBPLegend` that builds a plot legend and returns it to the caller as a plot object.

```
dfun=$DisplayFunction; If [$VersionNumber>=6.0, dfun=Print];
For [ip=1,ip<=1,ip++,  imgsiz=300;
    xy1={0,0}; xy2={imgsiz,0}; xy3={imgsiz,imgsiz}; xy4={0,imgsiz};
    a=b=imgsiz/2; xyleg={a+50,b+80}; If [ip==4, xyleg={a+50,b+125}];
    If [ip==6, xyleg={a-115,b-15}];
    fmax=100; If [ip==5, fmax=10000]; If [ip==6, fmax=1/3000];
    fmin=-fmax;  smax=1; If [ip==2,smax=1/2];
    mv=10;  If [ip==3,mv=2]; If [ip==4,mv=10];
    Print["fmin,fmax,xyleg,mv,smax=",{fmin,fmax,xyleg,mv,smax}];
    {preleg,plab}=CBPLegend[{fmin,fmax},{xyleg,mv},smax];
    Show[{Graphics[GrayLevel[0.9]],Graphics[Polygon[{xy1,xy2,xy3,xy4}]],
          Graphics[RGBColor[1,0,0]],Graphics[AbsoluteThickness[2]],
          Graphics[Line[{xy1,xy2,xy3,xy4,xy1}]]},preleg,plab,
          AspectRatio->Automatic,DisplayFunction->dfun];
    ];
```

FIGURE G.11. Test script for exercising `CBPLegend`, which produces the six plots of Figure G.12.

§G.2.5. Build Plot Legend: CBPLegend

Module CBPLegend generates a *plot legend*, which is a tabular visualization of the value-to-color mapping. The module source code is listed in Figure G.10. The module is invoked as

```
{preleg,plab}= CBPLegend[frange,xyleg,mv,smax]
```
(G.5)

The arguments are

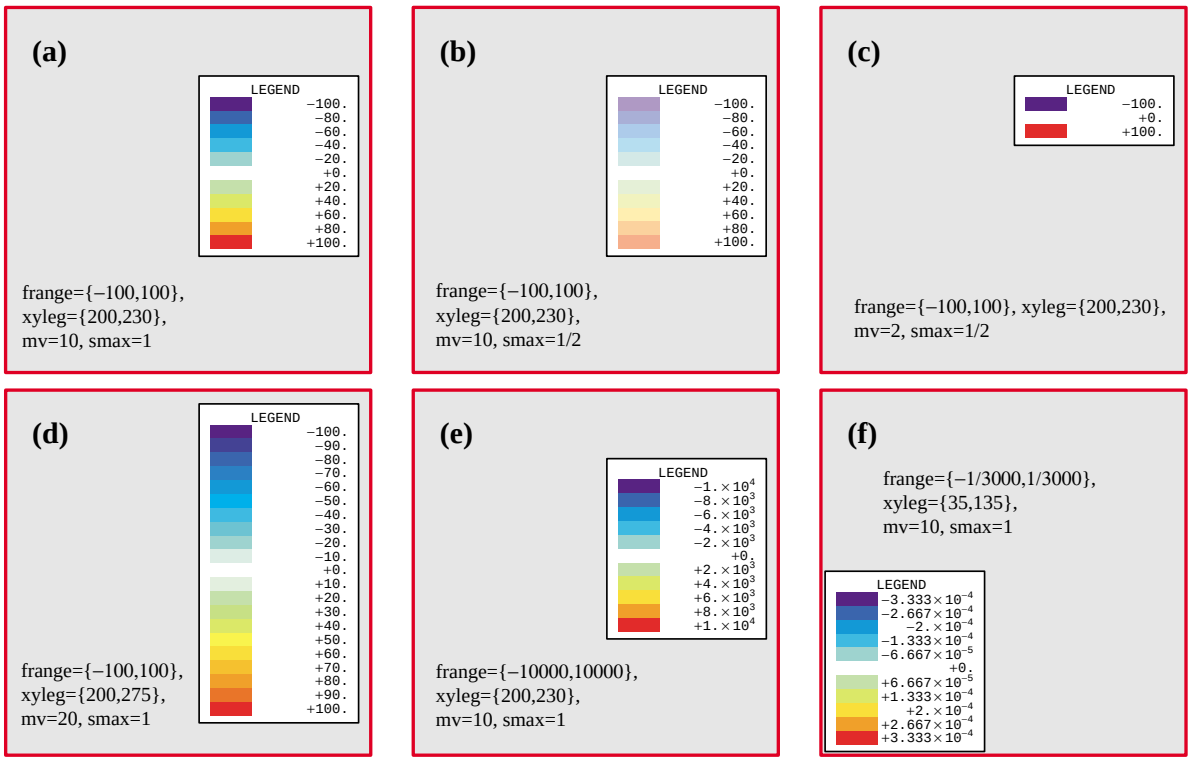

FIGURE G.12. Legend plots produced by the test script in Figure G.11

frange — A list { `fmin,fmax` } of minimum and maximum function values, respectively, which defines the plot range.

xyleg — The $\{x, y\}$ coordinates of a location that collocates the legend frame in the plot. See the source code for details. (The positioning logic is likely to be changed in the future, as it is currently confusing.)

mv — A positive integer that specifies the contour interval. values as `(fmax-fmin)/mv`. For example, if `mv=2` three values will appear in the legend table: `fmin`, `(fmin+fmax)/2`, and `fmax`. Note that `mv` is the same as the `ncint` input to the driver `CBPOver2DMeshMini` if texttt.ncint>0

smax — Maximum saturation value in color specification returned by `CBPColorMap`; see §G.2.6 for details. Normally 1.

The function returns { `preleg,plab` }, in which

preleg — Graphic commands to draw the legend table frame and title.

plab — Graphic commands to show colors alongside function values.

These graphic objects are inserted in the command plot stream by the driver program.

The legend module may be exercised by the plot script listed in Figure G.11. Running the script produces the plots shown in Figure G.12. These can be used to visualize various effects and options.

```
CBPColorMap[f_,frange_,smax_]:= Module[{fmin,fmax,fs,
   h,s,b=1,cs=25}, {fmin,fmax}=frange;
   If [Abs[fmax]<=0 || fmin>=fmax, (* White if void range *)
       Return[Graphics[Hue[0,0,1]] ]];
   If [f>fmax || f<fmin,      (* Black if f outside range *)
       Return[Graphics[Hue[0,0,0]] ]];
   fs=(fmax-f)/(fmax-fmin); h=0.64*fs;
   s=smax*(1+1/cs)*(1-1/(1+cs*(1-2*fs)^2));
   Return[Graphics[Hue[h,s,b]]]];
```

FIGURE G.13. Module `CBPColorMap` that maps a function value to a color defined through the built-in `Hue` graphics function.

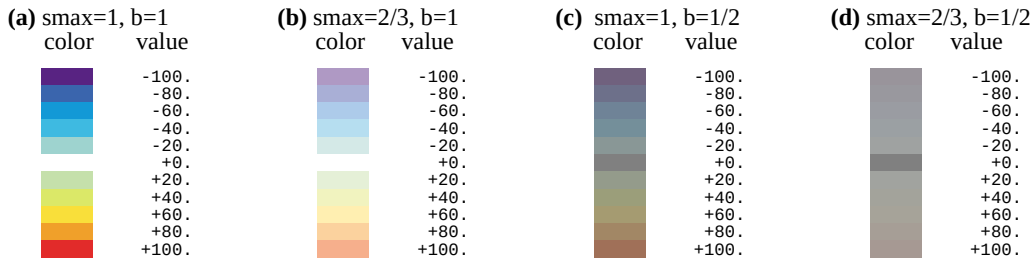

FIGURE G.14. Color palettes produced by calling `CBPColorMap` over the range `fmin=-100` through `fmax=+100` at `finc=+20` intervals. Brightness and saturation values as indicated in the Figure.

§G.2.6. Map Value to Color: CBPColorMap

Module `CBPColorMap`, listed in Figure G.13, maps a function value associated with a band to a color through the mapping scheme described below. The module is invoked by

```
color = CBPColorMap[f,frange,smax]
```
(G.6)

The arguments are:

f	Function value corresponding to color.
frange	A list { `fmin,fmax` } that defines the plot range.
smax	Maximum color saturation; normally 1 (see description below)

The module returns the color specification in terms of the `Hue` built-in function:

`Graphics[Hue[h,s,b]]`	The triple `h`, `s` and `b` specify the hue, saturation and brightness, respectively, of the returned color.

On entry `CBPColorMap` checks for void function range: $|\texttt{fmax}| \le 0$ or $\texttt{fmax} \le \texttt{fmin}$; if so it returns `Graphics[Hue[0,0,0]]`, which is white. If `f` is outside the range `[fmin,fmax]` it returns `[Hue[0,0,1]`, which is black. If both checks are passed, it computes the nondimensional function value `fs=(fmax-f)/(fmax-fmin)`, which varies from 0 for `f=fmax` to 1 for `f=fmin`. Hue and stauration are respectively set to `h=0.64*fs`, and `s=smax*(1+1/cs)(1-1/(1+cs*(1-2*fs)`^{2}`))` with `cs=25`. Brightness is internally preset to `b=1`.

The effect of these choices is illustrated in the color palettes shown in Figure G.14. If `f=fmin` so that `fs=0`, `h=0.64`, which with `s=1` gives deep blue, whereas if `f` reaches `fmax` so that `fs=0`, `h=0`, which along with `s=1` gives deep red. These extremes can be observed in the palette of Figure G.14(a).

For the average function value `f=(fmin+fmax)/2`, whence `fs=1/2`, one would like to get white. But white is not produced for any `h` as long as the saturation is nonzero The solution is to force the saturation to be zero if `fs=1/2`, which is done by the `s` function stated above. For `fs=0` and `fs=1` it yields `s=smax` as expected. The rational interpolation form and its coefficient `cs` were obtained by trial and error to get nice color variations in the vicinity of `f=0`, as can be observed in Figure G.14(a).

Setting the maximum saturation `smax` to less than 1 diffuses colors. This is evident in the palette of Figure G.14(b), produced with `smax=2/3`. Since this diffusivity has some uses, the value of `smax` can be set to the contour plotter from outside, overriding a default. On the other hand, moving brightness away from 1 is not a good idea since it mangles colors, as can be seen in the palettes of Figure G.14(c,d). For this reason `b=1` is internally set once and for all in the local variable list.

```
GenNodeCoordOverTrapezoid[xycorn_,nx_,ny_]:= Module[
   {k=0,i,j,dx,dy,numnod,N1,N2,N3,N4,xi,eta,
   xc1,xc2,xc3,xc4,yc1,yc2,yc3,yc4,x,y,elenod},
   numnod=(nx+1)*(ny+1); nodxyz=Table[0,{numnod}];
   {{xc1,yc1},{xc2,yc2},{xc3,yc3},{xc4,yc4}}=xycorn;
   For[i=1,i<=nx+1,i++, For [j=1,j<=ny+1,j++,
      xi=2*(i-1)/nx-1;  eta=2*(j-1)/ny-1;
      N1=(1-xi)*(1-eta)/4; N2=(1+xi)*(1-eta)/4;
      N3=(1+xi)*(1+eta)/4; N4=(1-xi)*(1+eta)/4;
      x=xc1*N1+xc2*N2+xc3*N3+xc4*N4;
      y=yc1*N1+yc2*N2+yc3*N3+yc4*N4;
      nodxyz[[++k]]={x,y};
      ]];
   Return[nodxyz]];

GenTrig3NodeNumbersOverTrapezoid[nx_,ny_,pat_]:= Module[{i,j,k,
   c1,c2,numele,enl},numele=({2,2,4,2}[[pat]])*nx*ny;
   enl=Table[{0,0,0},{numele}]; k=0;
   Do [Do [ c1=(ny+1)*(i-1)+j; c2=c1+ny+1;
       If [pat==1, enl[[++k]]={c1,c2,c1+1};
                   enl[[++k]]={c2+1,c1+1,c2}];
       If [pat==2, enl[[++k]]={c1,c2,c2+1};
                   enl[[++k]]={c2+1,c1+1,c1}];
       If [pat==3, enl[[++k]]={c1,c2,c1+1};
                   enl[[++k]]={c2+1,c1+1,c2};
                   enl[[++k]]={c1,c2,c2+1};
                   enl[[++k]]={c2+1,c1+1,c1}];
       If [pat==4, If [j<=ny/2,
                   enl[[++k]]={c1,c2,c2+1};
                   enl[[++k]]={c2+1,c1+1,c1},
                   enl[[++k]]={c1,c2,c1+1};
                   enl[[++k]]={c2+1,c1+1,c2}]],
   {j,1,ny}],{i,1,nx}]; Return[enl]];

GenQuad4NodeNumbersOverTrapezoid[nx_,ny_]:= Module[
   {k,i,j,c1,c2,numele,elenod},numele=nx*ny;
   elenod=Table[{0,0,0,0},{numele}]; k=0;
   For [i=1,i<=nx,i++, For [j=1,j<=ny,j++,
       c1=(ny+1)*(i-1)+j; c2=c1+ny+1;
       elenod[[++k]]={c1,c2,c2+1,c1+1}
       ]]; Return[elenod]];
```

FIGURE G.15. Mesh generation modules used by the `CBPOver2DMeshMini` tester listed in Figure G.16.

Remark G.1. Sometimes it is desirable to associate white with the zero value of the function, instead of the average. To do that, specify the function range as `-fmax,fmax`, in which `fmax` is the maximum absolute value. Obviously the average is zero. For example, suppose that the node function values span the range −85 to 26. Then set `fmin=-85` and `fmax=+85` in the inputs to the driver module.

```
nx=ny=8; scale=1; nodxyz=GenNodeCoordOverTrapezoid[{{0,0},{48,44},
      {48,60},{0,44}},nx,ny]; aspect=60/48;
elenod=GenTrig3NodeNumbersOverTrapezoid[nx,ny,1];
fun[x_,y_]:=N[(x-24)^2+(y-30)^2-200]*scale;
numele=Length[elenod]; numnod=Length[nodxyz]; fn=Table[0,{numnod}];
For[n=1,n<=numnod,n++, {xn,yn}=nodxyz[[n]]; fn[[n]]=fun[xn,yn]];
fmin=Min[fn]; fmax=Max[fn]; ncint=15; frain={fmin,fmax,ncint};
plotspecsleg={{{28,18},6},aspect,"Bandplot over Cook's Wing"};
plotspecsnoleg={{},aspect,"Bandplot over Cook's Wing"};
Timing[CBPOver2DMeshMini[nodxyz,elenod,eletyp,fn,frain,
  {True,False,False,True,True},plotspecsleg]]
Timing[CBPOver2DMeshMini[nodxyz,elenod,eletyp,fn,frain,
  {True,False,False,True,False},plotspecsleg]]
Timing[CBPOver2DMeshMini[nodxyz,elenod,eletyp,fn,frain,
  {False,False,False,True,True},plotspecsnoleg]]
Timing[CBPOver2DMeshMini[nodxyz,elenod,eletyp,fn,frain,
  {False,True,True,True,False},plotspecsnoleg]]
elenod=GenQuad4NodeNumbersOverTrapezoid[nx,ny];
Timing[CBPOver2DMeshMini[nodxyz,elenod,eletyp,fn,frain,
  {True,False,False,True,True},plotspecsleg]]
Timing[CBPOver2DMeshMini[nodxyz,elenod,eletyp,fn,frain,
  {True,False,False,True,False},plotspecsleg]]
Timing[CBPOver2DMeshMini[nodxyz,elenod,eletyp,fn,frain,
  {False,False,False,True,True},plotspecsnoleg]]
Timing[CBPOver2DMeshMini[nodxyz,elenod,eletyp,fn,frain,
  {False,True,True,True,False},plotspecsnoleg]]
```

FIGURE G.16. Test script for `CBPOver2DMeshMini`.

§G.2.7. Testing the Contour Band Plotter

The CBP plotter is exercised by a script that generates regular meshes of triangles and quadrilaterals over a planform known as "Cook's wing" in the FEM literature. It is a flat trapezoid with the corner x-y coordinates

```
{{0,0}, {48,44}, {48,60}, {0,44}}
```

Regular meshes with `nx` elements in the x direction and `ny` in the y direction are constructed by the auxiliary modules listed in Figure G.15. Module `GenNodeCoordOverTrapezoid` generates the node coordinates using an isoparametric mapping scheme, and returns that array as function value. Meshes of 3-node triangles and 4-node quadrilaterals are generated by modules `GenTrig3NodeNumbersOverTrapezoid` and `GenQuad4NodeNumbersOverTrapezoid`, respectively. These return the element nodelist array as function values.

The CBP tester script is listed in G.16. The node coordinates and element nodelists are placed in `nodxyz` and `elenod`, respectively. The function to be plotted is the quadratic polynomial

$$f(x, y) = (x - 24)^2 + (y - 30)^2 - 200.$$

whose $f = C$ lines are circles centered at $x = 24$ and $y = 30$. Meshes are generated with `nx=ny=8`. The function is evaluated at the nodes of the generated mesh and placed into the `fn` array. The

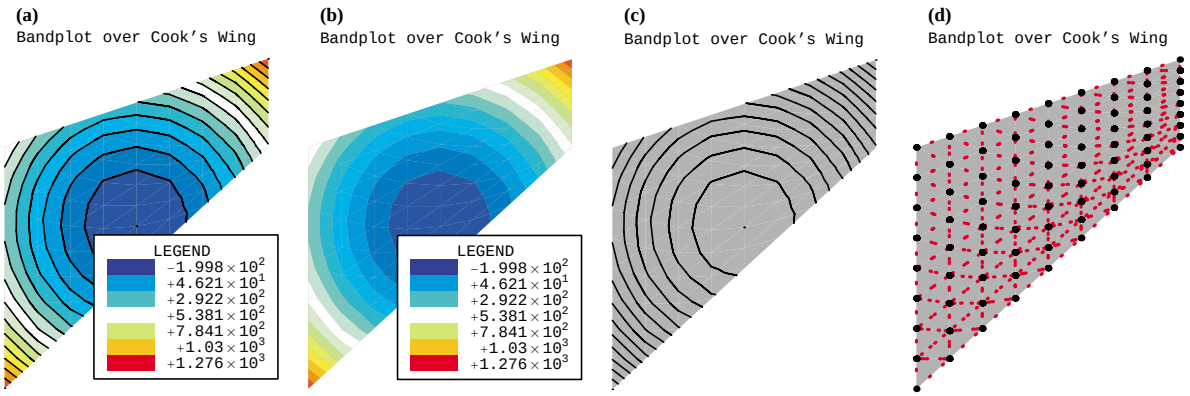

FIGURE G.17. Plots produced by the tester script of Figure G.16, over an 8×8 mesh of 3-node triangles.

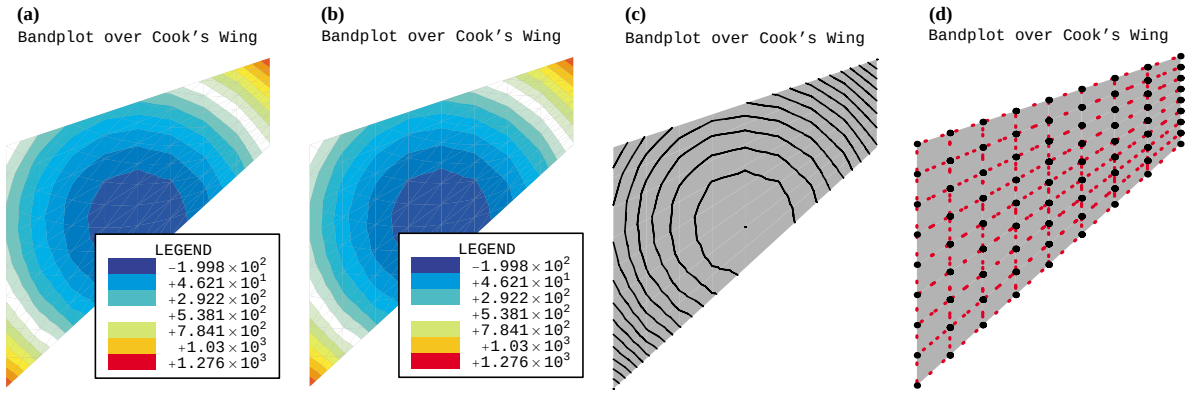

FIGURE G.18. Plots produced by the tester script of Figure G.16, over an 8×8 mesh of 4-node quadrilaterals.

minimum and maximum values are evaluated and placed into `fmin,fmax` }, respectively, to define the plot range. The number of contour intervals is `ncint=15`. Four plots are generated for each mesh. These are shown in G.17 and G.18 for the 3-node triangle and the 4-node quadrilaterals, respectively. The plots are:

(a) A CBP with contour lines drawn, plus legend.

(b) A CBP without contour lines, plus legend.

(c) A CLP without colors (grey background), and no legend.

(d) A mesh plot showing node locations and element sides (dashed lines), and no legend.

One unexpected feature is the appearance of the legend. Its frame looks much bigger in the figures than on the screen. (The figures were created by saving *Mathematica* plot cells to EPS files and postprocessed through Illustrator.) This unresolved anomaly may require some modification in the plot object creation logic.

§G.3. Genesis and Evolution of Plotting Utilities

Graphic utilities presented in this Appendix had origins in Fortran code written over five decades ago, not long after FEM exploded beyond aerospace applications. The author developed in 1965 a FEM contour plotter called PSI6PL for thesis work at Berkeley [203] under Ray Clough. That thesis dealt with 3- and 6-node triangles for membrane and plate bending. PSI6PL was written in Fortran IV. It was a classical ink-jet contour plotter, meaning that contour lines were produced as segments drawn with a ink pen on a Calcomp plotter.[4] Plot files were generated by a mainframe computer (at the time, an IBM 7094 that served the entire Berkeley campus) and stored on magnetic tape. The tape was later processed as an offline job, usually overnight.

PSI6PL was largely based on a similar program, called PSI3PL, written in Fortran II by Ed Wilson after completing his 1963 thesis [800], also under Ray Clough.[5] The main extension for PSI6PL was to allow 6-node triangles. PSI3PL was one of the original FEM graphics tools developed in an academic environment.[6]

Over the following decades PSI6PL underwent drastic changes. An alphanumeric version called CONTPP in which the plot was done on a line printer (a "printer plot") was developed while at Boeing [744]. This one allowed triangular and quadrilateral elements with or without midside nodes. A version called MESHPP that plotted a FEM mesh was published in 1972 [205]. During the 1970s that kind of machine independent plotters gained popularity because their output could be directed to a alphanumeric terminal such as DEC's VT100 series, popularized with the advent of minicomputers. Loss of plot quality was compensated by human time saved. Once satisfactory results were obtained, higher quality offline plots could be requested.

Although vector-display, interactive graphic terminals were available by the late 1960s, high prices and limited software kept them out of the mainstream. The turning point was reached in the early 1980s, when the emergence of pixel-display screens, personal computers, digital fonts, and laser printers brought high-quality graphics to a wider audience. The appearance in the 1990s of high level languages such as *Matlab* and *Mathematica*, which included color plotting as intrinsic feature, finally allowed FEM graphics to become platform independent.[7]

Successors of the original Fortran codes were written in *Mathematica* and used in FEM course development since the mid 1990s. They are described in various Sections throughout this Appendix.

---

[4] The Calcomp 565 drum plotter, introduced in 1959, was one of the first computer graphic output devices sold. The computer could control the rotation of an 11 inch wide drum within 0.01-in increments, as well as the horizontal movement of a pen holder over the drum. Paper rolls were 120-foot long. A pen was pressed by a spring against a paper scrolling over the drum. A solenoid could lift the pen off the paper. The arrangement allowed line drawings to be made under computer control. A metal bar above the take-up reel allowed the paper to be torn off and removed. The standard pen was a ball point, but liquid ink pens were also available for higher quality drawings. IBM sold the Calcomp 565 as the IBM 1627 for use in with its low-end scientific computers. Graphic output produced by batch jobs at high-end computers, such as the IBM 7094, was offloaded to tape and processed by the low-end one. Pictures of the Calcomp 565 (now in computer museums) may be seen in Google Images.

[5] Wilson's FEM thesis program was called PSI (developed 1959-63), for Plane Stress Iteration, because the DSM solution process was iterative in nature. It was the first fully automated FEM program based written in Fortran.

[6] Such programs seem to have been written at aerospace companies in the late 1950s, but for obvious reasons there is no clear historical trace.

[7] Fortran, like all low-level languages, never had platform independent graphics. It had to call machine-dependent libraries typically written in assembly language.