Matlab Code Analysis

Sungmin Cho

May 3, 2006

Contents

1 readpanel.m

1.1 Data format

There are two kinds of 'qif' file format.

- Q(Quadrature) format
- T(Trianglular) format

Q format

Q format is a format for plate shape, which comprises of four points to represent one rectangle.

```
Q SIZE VAL1 VAL2 ... VAL12
...
```

The line that starts without 'Q' is ignored. There are 12 numbers in one line, the three numbers builds one point as in (X, Y, Z), so there are four points in one line. So each line represents one plate(rectangle).

Following is an example of Q format file.

```
1  0 1mX1m single plate capacitor with 0.1m separation (n=3 e=1)
2  Q 1 6.66667e-01 6.66667e-01 ... 0.0e+00 6.66667e-01 1.0e+00 0.0e+00
3          ...
4  Q 1 6.66667e-01 6.66667e-01 ... 0.0e+00 6.66667e-01 1.0e+00 0.0e+00
5  *
```

T format

T format is a format for triangular shape, which comprises of three points to represent one rectangle.

```
T SIZE VAL1 VAL2 ... VAL9 A B C
...
```

There are 9 numbers in a line, so those three points represent one triangular shape.

There are three mover values for the representation.

A *Potential*

B $\frac{dPotential}{dn}$

C Type

Following is an example of T format file.

```
1  0 grid generated by mksphere -- 48 panels normal points outward
2  T 1 -0.000000 -0.0 1.0 ... -0.0 -0.7 0.7 0.0 0.0 0
3     ...
4  T 1 -0.707107 -0.0 0.7 ... -0.0 -0.0 1.0 0.0 0.0 0
```

1.2 Code Analysis

The function **readpanel** reads a panel from the source file **readpanel** reads one line in Q or T formmat and transform the line into a new data structure. And append the data structure at the 'List'. 'panelsize' is 4 for Q format and 3 form T format.

3 ⟨*readpanel* 3⟩≡

```
function [List] = readpanel(line,List,panelsize)
% Reads a panel with vertices as floating point numbers on a line
% Input
%  line <- line to be parsed, Q or T format
%  List <- Internal data structure
%  panelsize <- 3 for T format, 4 for Q format
```

⟨*scan the line* 4a⟩
⟨*Check if the line is Q format or T format* 4b⟩
⟨*Check the size of the list* 5⟩
⟨*Fill out the list* 6a⟩
⟨*Fill out the list for T format* 6b⟩
⟨*Fill out the values* 7⟩

Defines:
`readpanel`, never used.

To scan the line Matlab function **sscanf** is used. As a return 'stuff' stores ever value from the scanned result and 'cnt' stores the number of values that is read into the 'stuff'. Following is the example of **sscanf** function.

```
>> x = '1 2 3';
>> [a,b] = sscanf(x,'%d %d %d');
>> a
a =
     1
     2
     3
>> a(1)
ans =
     1
>> b
```

```
b =
     3
```

4a ⟨*scan the line* 4a⟩≡ (3)

```
[stuff,cnt]=sscanf(line,'%s %d %f %f %f %f %f %f %f %f %f %f %f %f %f %f');
```

As 'panelsize' is 4 for Q format and 3 for T format, the 'cnt' value can be used to check if the format is right or wrong.

4b ⟨*Check if the line is Q format or T format* 4b⟩≡ (3)

```
% Check if Q or T format is OK.
if (cnt == (2 + panelsize * 3))
  fastcap = 1;
  fastlap = 0;
elseif (cnt == (2 + 3 + panelsize * 3))
  fastcap = 0;
  fastlap = 1;
else
  disp('Format error in panel!');
  return;
end
```

'List' is a 3 dimensional matrix. The first two elements are 'row' and 'column' of the panel, and the last element is the index of the panel. The first row and the first column has the information of how many points the data has. For the Q format, it is 4, and for the T format, it is 3.

Following is an example of 'List' data structure of Q format with 3 data elements.

```
ans(:,:,1) =
    4.0000    1.0000         0
         0         0         0
    0.3333         0         0
    0.3333    0.3333         0
         0    0.3333         0

ans(:,:,2) =
    4.0000    1.0000         0
         0    0.3333         0
    0.3333    0.3333         0
    0.3333    0.6667         0
         0    0.6667         0
```

Following is an example of 'List' data structure of Q format with 3 data elements.

```
panels(:,:,1) =
    3.0000    1.0000         0
         0         0    1.0000
   -0.5774   -0.5774    0.5774
         0   -0.7071    0.7071
         0         0         0

panels(:,:,2) =
    3.0000    1.0000         0
         0   -0.7071    0.7071
    0.5774   -0.5774    0.5774
         0         0    1.0000
         0         0         0
```

5 ⟨*Check the size of the list* 5⟩≡ (3)

```
[rows,cols,numpanels] = size(List);
```

The first row of the new data is the information of the panel. The first column is panelsize(4 for Q format, 3 for T format), the second column is 'stuff(2)', which is the second element of the each line in the file. If the following example is read, '1' is the 'stuff(2)'.

```
Q 1 6.66667e-01 6.66667e-01 ... 0.0e+00 6.66667e-01 1.0e+00 0.0e+00
  ^
```

6a ⟨*Fill out the list* 6a⟩≡ (3)

```
% Add one data structuer
numpanels = numpanels +1;
List(1,1,numpanels) = panelsize;
List(1,2,numpanels) = stuff(2);
List(1,3,numpanels) = 0;
```

If the data is T format, the final three data should be stored in the last column.

```
panels(:,:,1) =
    3.0000    1.0000         0 <-- fastlap == 1, T format
         0         0    1.0000
   -0.5774   -0.5774    0.5774
         0   -0.7071    0.7071
         0         0         0 <-- panelsize + 2 = 5
```

6b ⟨*Fill out the list for T format* 6b⟩≡ (3)

```
% fastlap == 1 means T format
% panelsize + 2 is always 5
if(fastlap == 1)
  List(panelsize+2,1,numpanels) = stuff(cnt - 2);  % Potential
  List(panelsize+2,2,numpanels) = stuff(cnt - 1);  % d/dn Potential
  List(panelsize+2,3,numpanels) = stuff(cnt);      % Type
end;
```

Now all we have to do is just fill the data.

7 ⟨*Fill out the values* 7⟩≡ (3)

```
% stuff(index <- 3) is where the data starts.
index = 3;
for i=2:panelsize+1
  for j=1:3
    List(i,j,numpanels) = stuff(index);
    index = index + 1;
  end
end
```

2 readpanels.m

readpanels reads from a file T or Q format data line by line using the function **readpanel**

8a ⟨*readpanels* 8a⟩≡

```
function [panels] = readpanels(file)
% Read the file, line by line, and dispatch based on first character on line.
% Panel type, cond number, x1, y1, z1, x2, y2, z2, ... xn, yn, zn
% Panel type:
% Q means quadralateral
% T means triangle
% 0 or # or * means a comment
% Conductor number:
% An integer indicting a conductor number.
% Panel is stored as 3-D array
% [panel verts, cond num, 0]
% [vert 1 x,y,z]
% [vert 2 x,y,z]
% [vert 3 x,y,z]
%
% [vert n x,y,z]
% [potential, d/dn potential, type]
  ⟨set dummy 8b⟩
  ⟨file read 9a⟩
  ⟨remove dummy 9d⟩
```

Defines:
`readpanels`, never used.

Initialization code puts a dummy data in front of the first data. This is necessary for setting and initializing the 3D data.

8b ⟨*set dummy* 8b⟩≡ (8a)

```
panels(1,1,1) = 0.0;
```

Open the file and read the file line by line.

9a ⟨*file read* 9a⟩≡ (8a)

```
fid = fopen(file, 'r');
while 1
  line = fgetl(fid);
  % Break condition - no character (EOF)
  if ~ischar(line), break, end
  if length(line) ~= 0,
    if (strcmp(line(1),'Q') > 0) | (strcmp(line(1),'q') > 0)
      ⟨read Q format 9b⟩
    elseif(strcmp(line(1),'T') > 0) | (strcmp(line(1),'t') > 0)
      ⟨read T format 9c⟩
    % if the value is not '0','#','*', error
    elseif((strcmp(line(1),'0') == 0) & (strcmp(line(1),'#') == 0) & (strcmp(line(1),'*')
      'syntax error in input file'
      line
    end
  end
end

Reading Q and T format is same, the only difference is the last parameter.
```

9b ⟨*read Q format* 9b⟩≡ (9a)

```
panels = readpanel(line, panels, 4);
```

9c ⟨*read T format* 9c⟩≡ (9a)

```
panels = readpanel(line, panels, 3);
```

Now panels have the every information in 3D format. Finally, get rid of the dummy header panel.[1]

9d ⟨*remove dummy* 9d⟩≡ (8a)

```
% Get read of the dummy header panel.
[r,c,numpanels] = size(panels);
panels = panels(:,:,2:numpanels);
```

1) I think it's a clever idea to use a dummy data.

3 plotpanels.m

10a ⟨*plotpanels* 10a⟩≡

```
function plotpanels(panels)
% PLOTPANELS: plot the panels.
%
% Usage: plotpanels(panels)
%           Input - panels: 3D array of panel informations.
  ⟨get the size 10b⟩
  ⟨draw lines 10c⟩
```

Defines:
plotpanels, never used.

10b ⟨*get the size* 10b⟩≡ (10a)

```
% Find the number of panels.
[row,column,num_panels]=size(panels);
```

The first row and first column of 'panel' is the number of vertices in each panel. Using this information as 'num_verts', one can get line information.

10c ⟨*draw lines* 10c⟩≡ (10a)

```
figure(1), clf
for i=1:num_panels

    % Find the number of vertices in the panels.
    num_verts=panels(1,1,i);

    % Extract the edges.
    x=⟨getx 11a⟩
    y=⟨gety 11b⟩
    z=⟨getz 11c⟩

    ⟨draw a line 11d⟩

end
```

The vertice information of the panel starts at the second row and ends at the 'num_verts + 1' row. Together with that, one should add the first value again in order to draw the last line to the first line. The second parameter of 'panel' indicates 'x','y', or 'z'.

```
>> [panels(2:1+num_verts,1,i);panels(2,1,i)]
ans =
    0.7071
    0.5774
         0
    0.7071 <-- Repeats the first value
```

11a ⟨*getx* 11a⟩≡ (10c)

```
[panels(2:1+num_verts,1,i); panels(2,1,i)];
```

11b ⟨*gety* 11b⟩≡ (10c)

```
[panels(2:1+num_verts,2,i); panels(2,2,i)];
```

11c ⟨*getz* 11c⟩≡ (10c)

```
[panels(2:1+num_verts,3,i); panels(2,3,i)];
```

The matlab function **line**, can draw vector x, vector y, and vector z. The **line** selects x(1), y(1), and z(1) and x(2), y(2), and z(2) then draws a line. It iterates over and over again until it reaches the final data.

11d ⟨*draw a line* 11d⟩≡ (10c)

```
% Draw a panel.
line(x,y,z);
```

4 calcp.m

4.1 Code Analysis

12 ⟨*calcp* 12⟩≡

```
function [area,centroid,Z,fss,fds,fess,feds] = calcp(panel,evalpnts,directions)
% Matlab version of calcp, returns potential at evaluation point due
% to unit monopole and unit dipole uniformly distributed on a panel.
% Follows a left-hand rule (Clockwise ordered points has normal
% pointing up).
% panel -- vectors of panel vertices in rows of x,y,z (3 or 4 rows supported).
% evalpnts -- matrix of evaluation points, rows of x,y,z coordinates
% directions -- matrix of derivative directions, rows of x,y,z coordinates
% fss = the vector of potentials due to a monopole
% fds = the vector of potentials due to a panel normal dipole distribution
% area = panel area.
% centroid = panel centroid.
% Z = panel normal.
% fess = the derivative of the monopole potential at evalpnt along direction
% fess = the derivative of the dipole potential at evalpnt along direction
%

  ⟨check the input 13a⟩
  ⟨check the arguments 13b⟩
% The PANEL SETUP!!!!!!!!!***********************************************
  ⟨set sides 14b⟩
  ⟨calculate X,Y and Z 15⟩
  ⟨get the area 16a⟩
  ⟨normalize panel axies 16b⟩
  ⟨get the centroid points 17⟩
  ⟨check that panel is in the x-y panel 18a⟩
  ⟨Compute the contributions terms for each edge 18b⟩
% Done with the PANEL SETUP!!!!!!!!!***************************************
  ⟨Done with Setup, now loop through the evaluation points!! 19⟩
```

Uses area 16a, centroid 17, and Z 16b.

The 'panel' has the data structure for Q format as follows.

```
     0    0.3333         0
0.3333    0.3333         0
0.3333    0.6667         0
     0    0.6667         0
```

And T format as follows.

```
      0         0    1.0000
-0.5774   -0.5774    0.5774
      0   -0.7071    0.7071
```

13a ⟨*check the input* 13a⟩≡ (12)

```
% First check the input.
[verts, betterbethree] = size(panel);

if betterbethree ~= 3
  'wrong panel format: should be rows of x,y,z vectors!'
  return;
end

if (verts > 4) | (verts < 2)
  'wrong panel format: panel can only have 3 or 4 vertices!'
  return;
end
```

This is a clever trick for checking input arguments. If there is a argument, the two code blocks are run. If there is only one argument, only the first block is run.

13b ⟨*check the arguments* 13b⟩≡ (12) 14a▸

```
% If the number of input argument is '1'
% Check evaluation points
if(nargin > 1)
  [numevals, betterbethree] = size(evalpnts);

  if betterbethree ~= 3
    'wrong evaluation point format: should be rows of x,y,z vectors!'
    return;
  end
else
  numevals = 0;
end
```

14a ⟨*check the arguments* 13b⟩+≡ (12) ◁13b

```
% If the number of input argument is 2
% Check directions
if (nargin > 2)
  deriv = 1;
  [numdirections, betterbethree] = size(directions);

  if betterbethree ~= 3
    'wrong direction vector format: should be rows of x,y,z vectors!'
    return;
  end

  if (numdirections ~= 0) & (numdirections ~= numevals)
    'number of direction vectors does not match number of evaluation points!'
    return;
  end
else
  deriv = 0;
end
```

One can get sides and edgeLength by calculating panels. Figure 1 shows how to do it.

14b ⟨*set sides* 14b⟩≡ (12)

```
% Length of each side and the panel area.
for i=1:verts
  if(i < verts)
    side(i,:) = panel(i+1,:) - panel(i,:);
  else
    % Final Value
    side(i,:) = panel(1,:) - panel(i,:);
  end
  % edgeLength has the lenght of each sides.
  edgeLength(i) = norm(side(i,:));
end
```

Defines:
edgeLength, used in chunks 18b and 19.
side, used in chunk 19.
Uses area 16a.

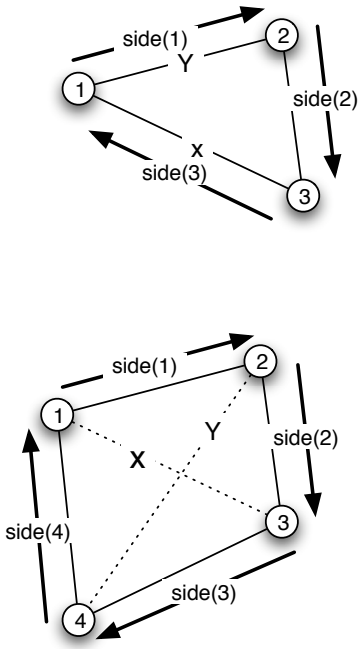

Figure 1: panels to get the X,Y, and Z

From the information in the panel, we can calculate X, Y and Z. The X is the difference between first and third value. If verts is 3, that is, with the T format, the Y is the difference between second and first value. With the Q format (verts == 4), the Y is the difference between fourth and second.

The figure 1 shows the result.

The Z can be calculated using outer product, $\vec{Z} = \vec{X} \times \vec{Y}$. **cross** matlab function is for making the vector.

15 ⟨*calculate X,Y and Z* 15⟩≡ (12)

```
% Calculate the panel coordinate system.
X = panel(3,:) - panel(1,:);
diagLength = norm(X);
if(verts == 3)
  Y = panel(2,:) - panel(1,:);
else
  Y = panel(2,:) - panel(4,:);
end

% Z-axis is normal to two diags.
Z = cross(X, Y);
```

Defines:
`diagLength`, used in chunks 17 and 18a.

Uses X 16b, Y 16b, and Z 16b.

By calculating the length of the Z, one can calculate the area.[2]

16a ⟨*get the area* 16a⟩≡ (12)

```
??? same rule for Q format and T format?
area = 0.5 * norm(Z);
```

Defines:
area, used in chunks 12, 14b, and 19.
Uses Z 16b.

Using the X,Y, and Z information, the new and normalized values can be obtained. Unit vector X, Y, and Z can be obtained.

16b ⟨*normalize panel axies* 16b⟩≡ (12)

```
% Normalize panel axises.
coord(3,:) = Z / norm(Z);
coord(1,:) = X / norm(X);
X = coord(1,:);
Z = coord(3,:);
coord(2,:) = cross(Z, X);
Y = coord(2,:);
```

Defines:
coord, used in chunks 17 and 22.
X, used in chunks 15 and 17.
Y, used in chunks 15 and 17.
Z, used in chunks 12, 15, and 16a.

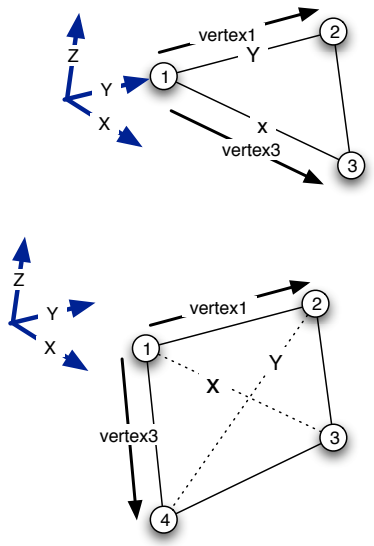

Figure 2: Getting centroid valule

With the information of unit vector X, Y, and Z, centroid values can be obtained. Figure 2 shows the diagram to get the centroid value. [3]

The figure 1 shows the result.

17 ⟨*get the centroid points* 17⟩≡ (12)

```
% Determine the centroid.
vertex1 = panel(2,:) - panel(1,:);
if(verts == 4)
  vertex3 = panel(4,:) - panel(1,:);
else
  vertex3 = panel(3,:) - panel(1,:);
end

% Project into the panel axes.
y1 = sum(vertex1 .* Y);
y3 = sum(vertex3 .* Y);
x1 = sum(vertex1 .* X);
x3 = sum(vertex3 .* X);
% y and z are orthogonal, so the third value is 0.
yc = (y1 + y3 + 0)/3.0;
% ??? don't understand this part.
xc = (diagLength + ((x1 * y1 - x3 * y3)/(y1 - y3)))/3.0;
```

2) T format works, but I think Q format doesn't work.
3) Needs more diagram.

```
% Compute the centroid.
centroid = panel(1,:) + xc * X + yc * Y;

% Put the corners in the newly defined coordinate system.
% ??? What's the rule?
% It looks like that we have a new panel named 'npanel'
for i=1:verts
  npanel(i,:) = (coord * (panel(i,:) - centroid).').';
end
```

Defines:
centroid, used in chunks 12 and 22.
npanel, used in chunks 18 and 19.
Uses coord 16b, diagLength 15, X 16b, and Y 16b.

18a ⟨*check that panel is in the x-y panel* 18a⟩≡ (12)

```
% Check that panel is in the x-y panel.
for i=1:verts
  if(abs(npanel(i,3)) > (1.0e-8 * diagLength))
    'Coordinate transform failure!!'
     npanel
     return;
  end;
end;
```

Uses diagLength 15 and npanel 17.

We have a new panel named 'npanel', with 'npanel', we can get 'ct' and 'st'.

18b ⟨*Compute the contributions terms for each edge* 18b⟩≡ (12)

```
% Compute the contributions terms for each edge.
for i=1:verts
  if (i==verts)
    next=1;
  else
    next=i+1;
  end;
  ct(i) = (npanel(next,1)-npanel(i,1))/edgeLength(i);
  st(i) = (npanel(next,2)-npanel(i,2))/edgeLength(i);
end
```

Defines:
ct, used in chunk 19.
st, used in chunk 19.
Uses edgeLength 14b and npanel 17.

19 ⟨*Done with Setup, now loop through the evaluation points!!* 19⟩≡ (12)

```
% Done with Setup, now loop through the evaluation points!!
for evalindex = 1:numevals
  ⟨Point eval pt in in new coordinate system and get the z-comp 22⟩
  % Once per vertex computation
  OK = 1;
  for i=1:verts
    xc=point(1)-npanel(i,1);
    yc=point(2)-npanel(i,2);
    zc=point(3)-npanel(i,3);
    xmxv(i)=xc;
    ymyv(i)=yc;
    fe(i)=xc*xc+zc*zc;
    r(i)=sqrt(yc*yc+fe(i));
    if (r(i) < (1.005*znabs))
      OK = 0;
    end;
    if(deriv == 1)
      xri(i) = xmxv(i)/r(i);
      yri(i) = ymyv(i)/r(i);
    end;
  end;

  % The potential and dipole contributions are made by summing up
  % a contribution from each edge
  fs=0;
  fd=0;
  if(deriv == 1)
    fsx = 0; fsy = 0;
    fdx = 0; fdy = 0; fdz = 0;
  end

  for i=1:verts
    if (i==verts)
      next=1;
    else
      next=i+1;
    end;
    % v is the projection of the eval-i edge on the perpend to the side-i:
    % Exploits the fact that corner points in panel coordinates.
    v=xmxv(i)*st(i) - ymyv(i)*ct(i);

    % arg == zero if eval on next-i edge, but then v = 0. %
    arg=(r(i)+r(next)-edgeLength(i))/(r(i)+r(next)+edgeLength(i));
    if(arg == 0)
      'in calcp'
```

```
  keyboard;
end
fln = -log(arg);
if (arg>0.0)
  fs = fs + v * fln;
end;
if ( deriv == 1 )
  if ( arg > 0.0 )
    fac = (r(i)+r(next)-edgeLength(i)) * (r(i)+r(next)+edgeLength(i));
    fac = v*(edgeLength(i)+ edgeLength(i))/fac;
    fsx = fsx + (fln*st(i) - fac*(xri(i) + xri(next)));
    fsy = fsy - (fln*ct(i) + fac*(yri(i) + yri(next)));
    fdz = fdz - (fac*( 1.0/r(i) + 1.0/r(next)));
  end
end

% OK means eval not near a vertex normal, use Hess-Smith:
if (OK == 1)
  s1=v*r(i);
  c1=znabs*(xmxv(i)*ct(i)+ymyv(i)*st(i));
  s2=v*r(next);
  c2=znabs*(xmxv(next)*ct(i)+ymyv(next)*st(i));
else % Near a vertex normal, use Newman
  s1=(fe(i)*st(i))-(xmxv(i)*ymyv(i)*ct(i));
  c1=znabs*r(i)*ct(i);
  s2=(fe(next)*st(i))-(xmxv(next)*ymyv(next)*ct(i));
  c2=znabs*r(next)*ct(i);
end;

s12=(s1*c2)-(s2*c1);
c12=(c1*c2)+(s1*s2);
val=atan2(s12, c12);
fd=fd+val;
if (deriv == 1)
  u1   = xmxv(i)*ct(i) + ymyv(i)*st(i);
  u2   = xmxv(next)*ct(i)+ymyv(next)*st(i);
  if (OK == 0) % Near a vertex normal.
    rr  = r(i)*r(i);
    fh1 = xmxv(i)*ymyv(i);
    fh2 = xmxv(next)*ymyv(next);
    fac = c1/((c1*c1+s1*s1)*rr );
    if(zn < 0.0)
      fac = -1.0 * fac;
    end
    fdx = fdx + ((rr*v+fh1*u1)*fac);
    fdy = fdy - (fe(i)*u1*fac);
```

```
        rr  = r(next)*r(next);
        fac = c2/((c2*c2+s2*s2)*rr);
        if(zn < 0.0)
          fac = -1.0 * fac;
        end
        fdx = fdx - ((rr*v+fh2*u2)*fac);
        fdy = fdy + fe(next)*u2*fac;
      else
        fac = zn/(c1*c1+s1*s1);
        fdx = fdx + (u1*v*xri(i)+r(i)*ymyv(i))*fac;
        fdy = fdy + (u1*v*yri(i)-r(i)*xmxv(i))*fac;
        fac = zn/(c2*c2+s2*s2);
        fdx = fdx - ((u2*v*xri(next)+r(next)*ymyv(next))*fac);
        fdy = fdy - ((u2*v*yri(next)-r(next)*xmxv(next))*fac);
      end
    end;
  end;

  if (fd<0.0)
    fd = fd + 2*pi;
  end;
  if (zn < 0)
    fd=fd*(-1.0);
  end;

  fs=fs-zn*fd;

  if(deriv == 1 )
    fsx = fsx - zn*fdx;
    fsy = fsy - zn*fdy;
    fes = nrm(1)*fsx + nrm(2)*fsy - nrm(3)*fd;
    fed = nrm(1)*fdx + nrm(2)*fdy + nrm(3)*fdz;
  end

  % No area normalization!
  fss(evalindex) = fs;
  fds(evalindex) = fd;
  if(deriv == 1)
    fess(evalindex) = fes;
    feds(evalindex) = fed;
  end;
end;
```

Uses area 16a, ct 18b, edgeLength 14b, npanel 17, side 14b, and st 18b.

22 ⟨*Point eval pt in in new coordinate system and get the z-comp* 22⟩≡ (19)

```
% Point eval pt in in new coordinate system and get the z-comp.
point = (coord * ((evalpnts(evalindex,:) - centroid).')).';
if(deriv == 1)
  nrm = (coord * directions(evalindex,:).').';
end
zn=point(3);
znabs=abs(zn);
evalDistance = norm(point);
```

Uses centroid 17 and coord 16b.

5 calccap.m

calccap calculates capacitance of the panels using collocation method.

23a ⟨*calccap* 23a⟩≡
```
function [Creal,matrix] = calccap(infile)
  ⟨set the constant 23b⟩
  ⟨Read in the panels 23c⟩
  ⟨Compute centroid, normals, and areas 23d⟩
  ⟨Generate the collocation matrix 23e⟩
  ⟨Generate RHS, MATRIX and solve them 23f⟩
  ⟨Integrate(Sum) to get the result 24⟩
```
Uses `Creal` 24.

23b ⟨*set the constant* 23b⟩≡ (23a)
```
% Permitivity of free space.
E_0 = 8.854187818E-12;
```
Defines:
`E_0`, used in chunk 24.

23c ⟨*Read in the panels* 23c⟩≡ (23a)
```
% Read in the panels
[panels] = readpanels(infile);
done = 'read input file'
```

Use the **gencolloc** to calculate the centroids, normals, and areas.

23d ⟨*Compute centroid, normals, and areas* 23d⟩≡ (23a)
```
% Compute the panel centroids and areas (don't need the normals).
[centroids, normals, areas] = gencolloc(panels);
done = 'generated collocation points'
```

Now, we have centroids, using this information, one can get the collocation matrix. It's interesting that the same **calcp** is used to get the `matrix`.

23e ⟨*Generate the collocation matrix* 23e⟩≡ (23a)
```
% Generate the collocation matrix
[matrix] = collocation(panels,centroids);
done = 'generated matrix'
```

Now we can solve the matrix to get the ω, which is $Q = \omega \times AREA$.

23f ⟨*Generate RHS, MATRIX and solve them* 23f⟩≡ (23a)
```
% Create the rhs
  [r,c] = size(matrix);
  rhs = ones(r,1);

% Solve for the charge density vector
 q = matrix \ rhs;
done = 'solved for charge'
```

Using the equation $Q = CV$, and as the boundary condition of V as 1. We know that $Q = C$, so all we have to do is integrate or sum every Q.

After that, we multiply by the value $4\pi\epsilon$ to get the capacitance.[4]

24 ⟨*Integrate(Sum) to get the result* 24⟩≡ (23a)

```
% Integrate the charge over the surface to compute the capacitance.
% But first multiply the charge density by the panel area.
  q = q .* areas.';
  C = sum(q);

% Scale the capacitance by the free space dielectric permitivity and 4pi.
Creal = E_0 * 4 * pi * C;
```

Defines:
`Creal`, used in chunk 23a.
Uses `E_0` 23b.

4) Why $4\pi\epsilon$?

6 collocation.m

collocation uses the **calcp** in order to get the 'fss'. The 'fss' value is just the collocation matrix.

25 ⟨*collocation* 25⟩≡

```
function matrix = collocation(panels,centroids)
% Fills in matrix relating panel charges to collocation point potentials.
% Panel is stored as 3-D array
% [panel verts, cond num, 0]
% [vert 1 x,y,z]
% [vert 2 x,y,z]
% [vert 3 x,y,z]
%
% [vert verts x,y,z]
% Centroids stored as a matrix with 3 columns.
% [vert 1 x,y,z]
% [vert 2 x,y,z]
% [vert 3 x,y,z]
%
% Setting up the collocation matrix
 [rows,cols,numpanels] = size(panels);
 for i=1:numpanels
   numverts = panels(1,1,i);
   panel = panels(2:numverts+1,:,i);
   [area, collocpt, Z, fss] = calcp(panel, centroids);
   matrix(:,i) = fss';
 end
```

Defines:
collocation, never used.

7 List of code chunks

8 Index

Underlined indices denote definitions; regular indices denote uses.