7 Appendix

7.1 Algorithms

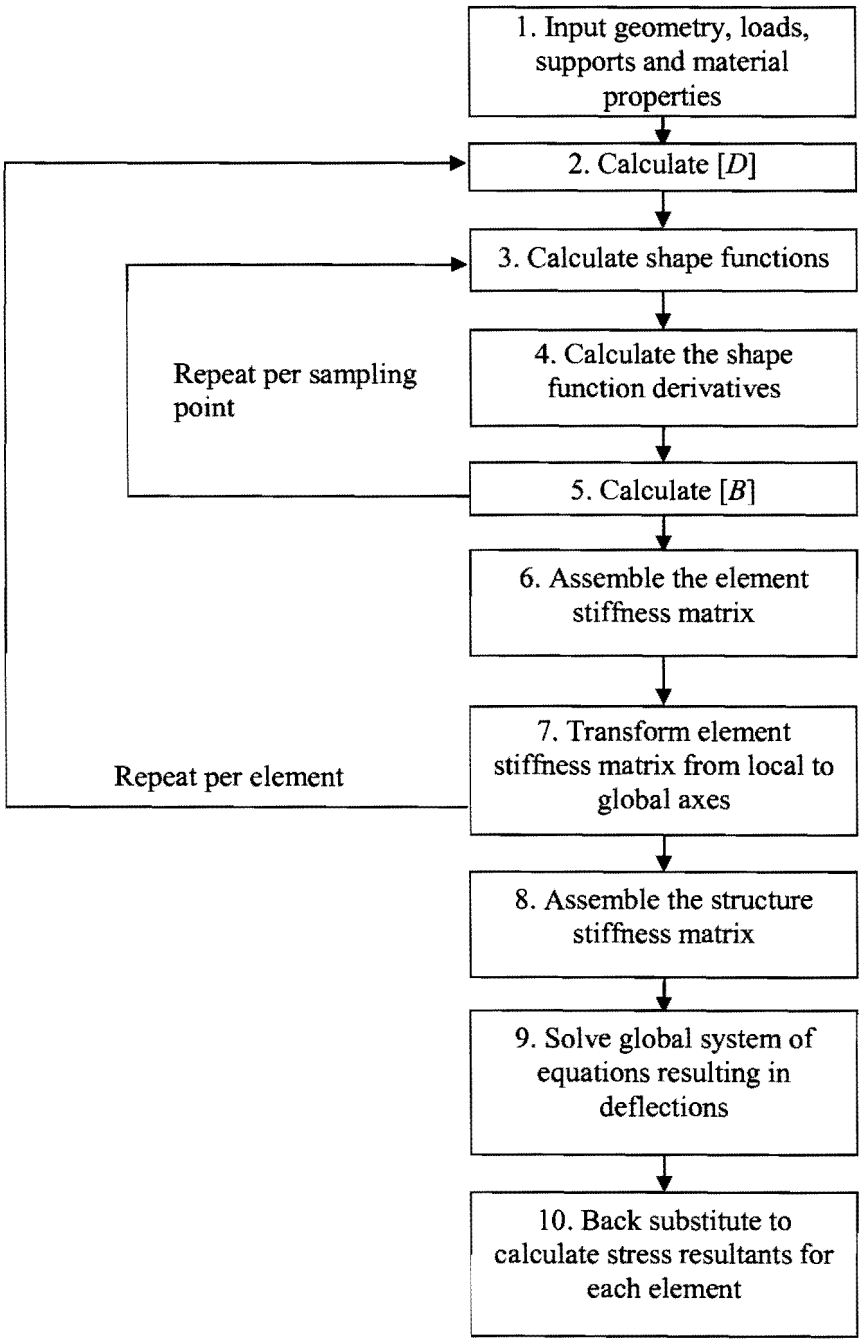

Figure 7-1: Linear finite element analysis algorithm

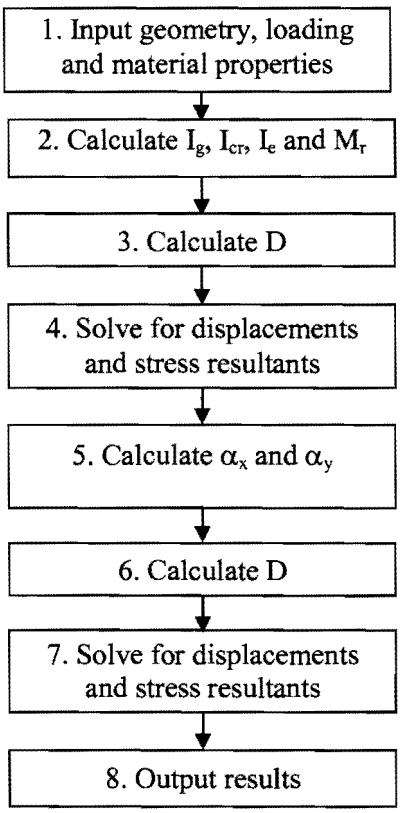

Figure 7-2: Polak's tension stiffening algorithm

Note: The calculation of [*D*] follows the procedure described in section 2.2.2.

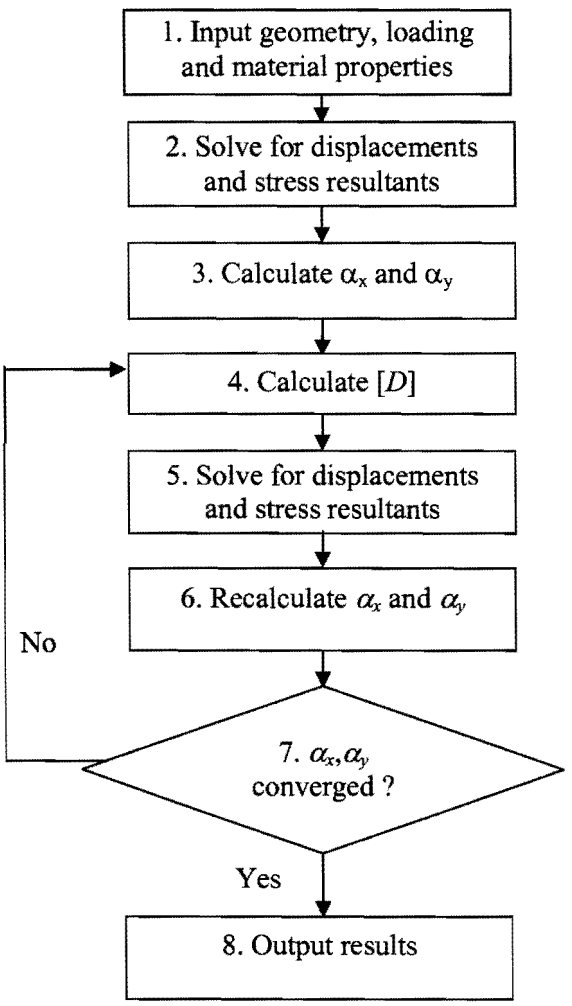

Figure 7-3: Modification of Polak's algorithm

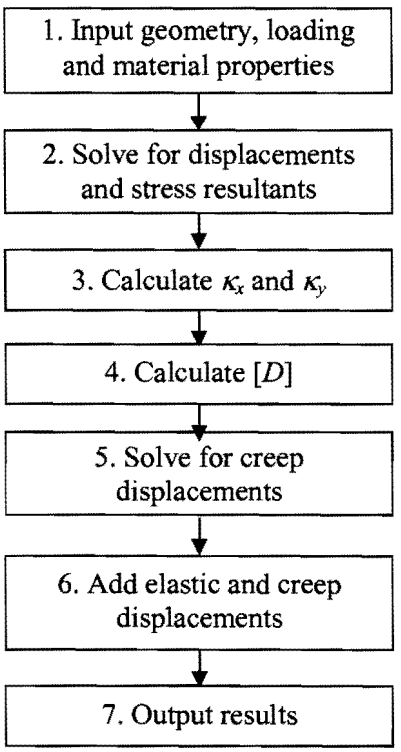

Figure 7-4: Creep analysis algorithm

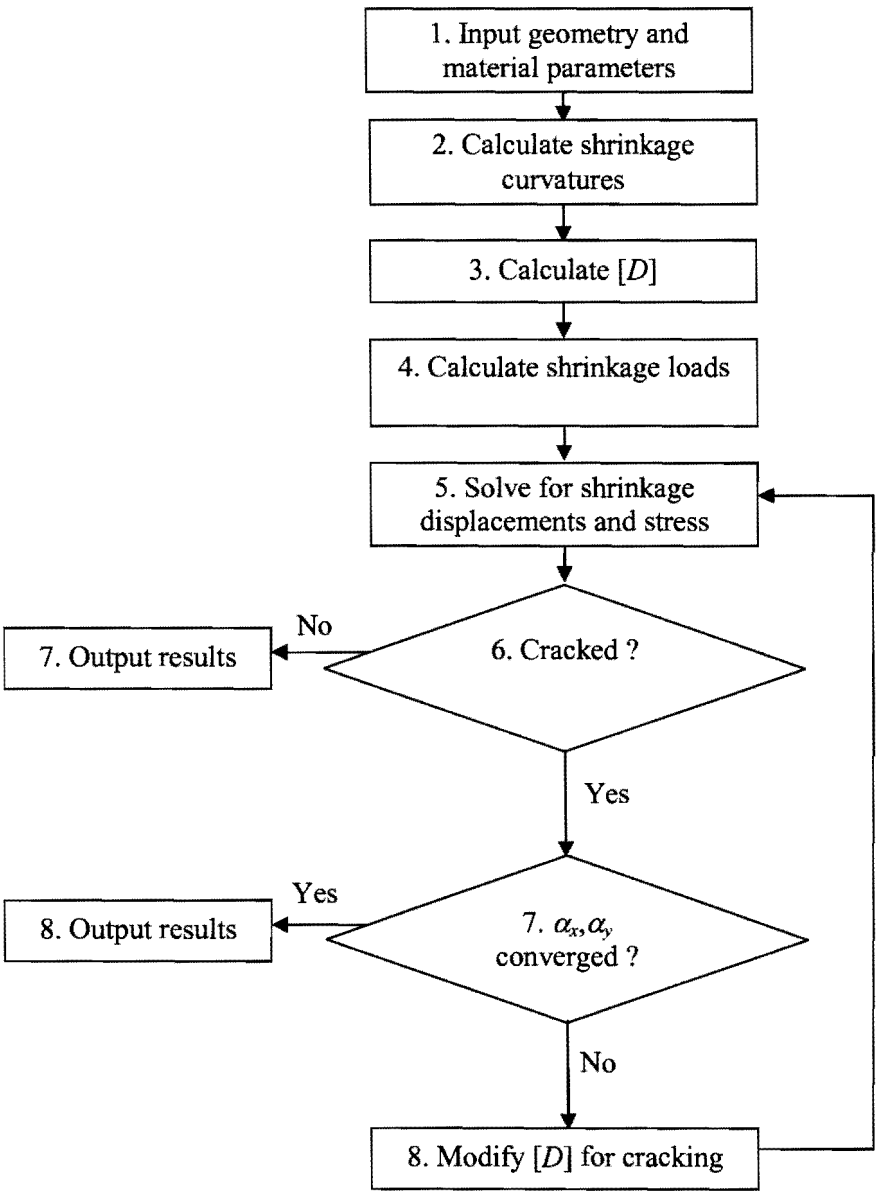

Figure 7-5: Shrinkage analysis algorithm

7.2 Unit Mindlin code listing

```
unit Mindlin;

interface

type

  TElement = class
    public
      //Public member variables
      Number       : Integer;                          //Element index in global array
      Es           : Double;                           //Young's modulus of reinforcement
      E            : Double;                           //Young's modulus of concrete
      Pois         : Double;                           //Poisson's ratio
      h            : Double;                           //Element depth
      G            : Double;                           //Shear modulus of concrete
      NumIntOrder  : Integer;                          //Order of numerical integration
      phi          : double;                           //Creep factor

      NodesCoord   : array[1..3,1..8] of Double;       //Nodal point coordinates
      NodesNum     : array[1..8] of Integer;           //Global node numbers

      ElStiffp     : array[1..24,1..24] of Double;     //Element stiffness matrix - plate
      ElStiffm     : array[1..16,1..16] of Double;     //Element stiffness matrix - membrane
      ElStiff      : array[1..40,1..40] of Double;     //Superposed Element stiffness matrix

      SRes         : array[1..4,1..8] of Double;       //Stress resultants
      SResM        : array[1..8,1..8] of Double;       //Smoothed stress resultants
      Def          : array[1..5,1..8] of Double;       //Nodal deflections

      GaussDBp     : array[1..5,1..24,1..4] of Double; //DxB per sampling point - plate
      GaussDBm     : array[1..3,1..16,1..4] of Double; //DxB per sampling point – membrane

      ElLoad       : array[1..24] of Double;           //Element load vector
      UDL          : Double;                           //UDL on element

      Mxavg, Myavg         : Double;                   //Average moments for crack analysis
      alphax, alphay       : Double;                   //Crack modification factors
      ksix, ksiy           : Double;                   //Bilinear crack modification factors
      alphaOldx, alphaOldy : Double;                   //Previous crack modification factors

      G1, G2, G3          : Double;                    //Orthotropic shear moduli
      Ex, Ey              : Double;                    //Orthotropic Young's moduli
      Poisx, Poisy        : Double;                    //Orthotropic Poisson's ratios

      Astx, Asty          : Double;                    //Area of tension steel mm²/m
      dtx, dty            : Double;                    //Effective depth of tension steel
      rocx, rocy          : Double;                    //Steel percentages, compression
      rox, roy            : Double;                    //Steel percentages, tension

      Igx, Igy            : Double;                    //Gross moments of inertia
      Iex, Iey            : Double;                    //Effective moments of inertia
      IeOldx, IeOldy      : Double;                    //Previous effective moments of inertia
      Icx, Icy            : Double;                    //Cracked moments of inertia
      Mcrx, Mcry          : Double;                    //Cracking moments
      Crackedx, Crackedy  : Boolean;                   //True/False crack flags

      Cx, Cy, Cxy         : Double;                    //Curvature variables
      xg, yg              : Double;                    //NA depth, gross, transformed
      xc, yc              : Double;                    //NA depth, cracked, transformed

      kappaX, kappaY      : Double;                    //Creep modification factors
      ecs                 : Double;                    //Free shrinkage strain
      MxShrink, MyShrink : double;                     //Shrinkage forces
```

```
    //Public procedures and functions
    Constructor Create(AOwner:TObject);   //Object creation - override
    Destructor Destroy; override;         //Object destruction - override

    Procedure CalcBp;                     //Calculates the plate strain matrix
    Procedure CalcBm;                     //Calculates the membrane strain matrix

    Procedure CalcDp;                     //Calculates the plate elastic matrix
    Procedure CalcDm;                     //Calculates the membrane elastic matrix

    Procedure CalcDBp;                    //Calculates product of B x D - plate
    Procedure CalcDBm;                    //Calculates product of B x D – membrane

    Procedure CalcJ( var JDet : Double;   //Calculates the jacobian matrix and its inverse
                     GNum     : Integer );
    Procedure CalcShape(u, v : Double);   //Calculates shape funtions and derivatives

    Procedure CalcStiffp;                 //Calculates the plate element stiffness matrix
    Procedure CalcStiffm;                 //Calculates the membrane element stiffness matrix
    Procedure CalcStiff;                  //Calculates the total element stiffness matrix

    Procedure SetupNumInt;                          //Sets up numerical integration
    Procedure CalcUDLoad;                           //Reduce UDL to nodal loads
    Procedure CalcShrinkLoad( CurvX : Double,       //Calculate shrinkage loads
                              CurvY : Double,
                              CurvXY: Double,
                              ShearX: Double,
                              ShearY: Double
    Procedure CalcAvgs;                             //Calculate average moments
    Procedure CalcModFactors(CrackType:integer);    //Calculate crack modification factors
    Procedure CalcCurvatures;                       //Calculate curvatures from deflections
    Procedure CalcCreepFactors;                     //Calculate creep modification factors
    Procedure InitInertia;                          //Initialize moments of inertia vars
    Procedure CalcShrinkageCurvatures;              //Calculate shrinkage curvatures

  private
    //Private member variables
    Bp      : array[1..5,1..24] of Double; //Strain matrix - plate
    Bm      : array[1..3,1..16] of Double; //Strain matrix - membrane

    Dp      : array[1..5,1..5]  of Double; //Elasticity matrix - plate
    Dm      : array[1..3,1..3]  of Double; //Elasticity matrix - membrane

    DBp     : array[1..5,1..24] of Double; //Product of D x B - plate
    DBm     : array[1..3,1..16] of Double; //Product of D x B – membrane

    J       : array[1..2,1..2]  of Double; //Jacobian matrix
    JI      : array[1..2,1..2]  of Double; //Jacobian matrix inverse
    GP      : array[1..2,1..4]  of Double; //Sampling point coordinates

    SFunc   : array[1..8]       of Double; //Shape funtions per node;
    SFDeriv: array[1..3,1..8]  of Double; //Shape function derivatives per node
    CDeriv : array[1..3,1..8]  of Double; //Cartesian shape function derivatives per node

    GaussPos : array[1..2]      of Double; //Sampling point position
    GaussWgt : array[1..2]      of Double; //Sampling point weighting factor

    calcForCreep : Boolean;                //True/False creep analysis flag
  published

end;
```

```
implementation

uses
  DTools, Math;

////////////////////////////////////////////////////////
// Object creator - simply initializes a few variables //
////////////////////////////////////////////////////////

Constructor TElement.Create(AOwner:TObject);
begin
  inherited Create;

  Crackedx := False;
  Crackedy := False;
  calcForCreep := False;
  InitInertia;
end;

////////////////////////////////////////////////
// Object destructor - calls default destructor //
////////////////////////////////////////////////

Destructor TElement.Destroy;
begin
  inherited Destroy;
end;

//////////////////////////////////////
// Calculates the plate strain matrix //
//////////////////////////////////////

Procedure TElement.CalcBp;
var
  iC,jC,kC : Integer;  //Loop counters

begin

  //Zero all matrix entries
  for iC := 1 to 5 do
    for jC := 1 to 3 do
      Bp[iC,jC] := 0;

  //Calculate the B matrix
  jC := 0;
  for iC := 1 to 8 do
  begin
    kC := jC + 1;
    Bp[4,kC] := CDeriv[1,iC];
    Bp[5,kC] := CDeriv[2,iC];
    kC := kC + 1;
    jC := kC + 1;
    Bp[1,kC] := -CDeriv[1,iC];
    Bp[3,kC] := -CDeriv[2,iC];
    Bp[4,kC] := -SFunc[iC];
    Bp[2,jC] := -CDeriv[2,iC];
    Bp[3,jC] := -CDeriv[1,iC];
    Bp[5,jC] := -SFunc[iC];
  end;

end;
```

```
////////////////////////////////////////////
// Calculates the membrane strain matrix //
////////////////////////////////////////////

Procedure TElement.CalcBm;
var
  iC,jC,kC : Integer;  //Loop counters

begin

  //Zero all matrix entries
  for iC := 1 to 3 do
    for jC := 1 to 2 do
      Bm[iC,jC] := 0;

  //Calculate the B matrix
  jC := 0;
  for iC := 1 to 8 do
  begin
    kC := jC + 1;
    jC := kC + 1;
    Bm[1,kC] := CDeriv[1,iC];
    Bm[1,jC] := 0;
    Bm[2,kC] := 0;
    Bm[2,jC] := CDeriv[2,iC];
    Bm[3,kC] := CDeriv[2,iC];
    Bm[3,jC] := CDeriv[1,iC];
  end;

end;

////////////////////////////////////////////
// Calculates the plate elasticity matrix //
////////////////////////////////////////////

Procedure TElement.CalcDp;
var
  FactorM, FactorV : Double;   //Temporary storage variables
  iC, jC           : Integer;  //Loop counters

begin

  //Zero all matrix entries
  for iC := 1 to 5 do
    for jC := 1 to 5 do
      Dp[iC,jC] := 0;

  if not (CalcForCreep) then
  begin
    //Do not use the creep modification factors
    if not (CrackedX or CrackedY) then
    begin
      //Calculate the gross D Matrix
      FactorM := E*power(h,3)/(12*(1-power(Pois,2)));
      FactorV := G*h/2.4;

      Dp[1,1] := FactorM;
      Dp[1,2] := Pois*FactorM;
      Dp[2,1] := Pois*FactorM;
      Dp[2,2] := FactorM;
      Dp[3,3] := G*pow(h,3)/12;
      Dp[4,4] := FactorV;
      Dp[5,5] := FactorV;
    end
    else
```

```
    begin
      //Calculate the cracked D Matrix
      Ex := E*alphax; Ey := E*alphay;
      Poisx := Pois*alphax; Poisy := Pois*alphay;
      G1 := G*alphax*alphay; G2 := G*alphax; G3 := G*alphay;


      Dp[1,1] := Ex*power(h,3)/(12*(1-Poisx*Poisy));
      Dp[1,2] := Poisx*Ey*power(h,3)/(12*(1-Poisx*Poisy));
      Dp[2,1] := Poisy*Ex*power(h,3)/(12*(1-Poisx*Poisy));
      Dp[2,2] := Ey*power(h,3)/(12*(1-Poisx*Poisy));
      Dp[3,3] := G1*power(h,3)/12;
      Dp[4,4] := G2*h;
      Dp[5,5] := G3*h;
    end;
  end
  else
  begin
    //Use the creep modification factors
    Ex := alphax*E/(phi*kappaX); Ey := alphay*E/(phi*kappaY);
    G1 := alphax*alphay*G/(phi*kappaX*kappaY);
    G2 := alphax*G/(phi*kappaX);
    G3 := alphay*G/(phi*kappaY);

    Dp[1,1] := Ex*power(h,3)/(12*(1-Pois*Pois));
    Dp[1,2] := Pois*Ey*power(h,3)/(12*(1-Pois*Pois));
    Dp[2,1] := Pois*Ex*power(h,3)/(12*(1-Pois*Pois));
    Dp[2,2] := Ey*power(h,3)/(12*(1-Pois*Pois));
    Dp[3,3] := G1*power(h,3)/12;
    Dp[4,4] := G2*h;
    Dp[5,5] := G3*h;
  end;

end;

////////////////////////////////////////////////
// Calculates the membrane elasticity matrix  //
//                                            //
// Note that no modifications have been made to //
// account for cracking, creep and shrinkage  //
////////////////////////////////////////////////

Procedure TElement.CalcDm;
var
  Constant : Double;   //Temporary storgare variable
  iC, jC   : Integer;  //Loop counters
begin

  //Zero all matrix entries
  for iC := 1 to 3 do
    for jC := 1 to 3 do
      Dm[iC,jC] := 0;

  Constant := E/(1-pow(Pois,2));
  Dm[1,1] := Constant;
  Dm[1,2] := Pois*Constant;
  Dm[2,1] := Pois*Constant;
  Dm[2,2] := Constant;
  Dm[3,3] := (1-Pois)/2*Constant;

end;
```

```
////////////////////////////////////////////////////////
// Calculates the product of B and D for re-use (plate) //
////////////////////////////////////////////////////////

Procedure TElement.CalcDBp;
var
  iC,jC,kC : Integer;  //Loop counter

begin

  //Calculate B x D
  for iC := 1 to 5 do
  begin
    for jC := 1 to 24 do
    begin
      DBp[iC,jC] := 0;
      for kC := 1 to 5 do
        DBp[iC,jC] := DBp[iC,jC] + Dp[iC,kC]*Bp[kC,jC];
    end;
  end;

end;

////////////////////////////////////////////////////////////
// Calculates the product of B and D for re-use (membrane) //
////////////////////////////////////////////////////////////

Procedure TElement.CalcDBm;
var
  iC,jC,kC : Integer;  //Loop counters
begin

  //Calculate B x D
  for iC := 1 to 3 do
  begin
    for jC := 1 to 16 do
    begin
      DBm[iC,jC] := 0;
      for kC := 1 to 3 do
        DBm[iC,jC] := DBm[iC,jC] + Dm[iC,kC]*Bm[kC,jC];
    end;
  end;

end;

////////////////////////////////////////////////////////
// Calculates the the Jacobian matrix and its inverse //
////////////////////////////////////////////////////////

Procedure TElement.CalcJ(var JDet : Double; GNum : Integer);
var
  iC,jC,kC : Integer;  //Loop counters

begin

  //Zero all matrix entries
  for iC := 1 to 2 do
    for jC := 1 to 2 do
    begin
      JI[iC,jC] := 0;
    end;

  //Calculate Gaussian point coordinates
  for iC := 1 to 2 do
  begin
    GP[iC,GNum] := 0;
    for jC := 1 to 8 do
    begin
      GP[iC,GNum] := GP[iC,GNum] + NodesCoord[iC,jC]*SFunc[jC];
```

```
    end;
  end;

  //Calculate the Jacobian matrix
  for iC := 1 to 2 do
  begin
    for jC := 1 to 2 do
    begin
      J[iC,jC] :=0;
      for kC := 1 to 8 do
      begin
        J[iC,jC] := J[iC,jC] + SFDeriv[iC,kC]*NodesCoord[jC,kC];
      end;
    end;
  end;

  //Calculate determinant and inverse of the Jacobian
  JDet := J[1,1]*J[2,2]-J[1,2]*J[2,1];
  JI[1,1] := J[2,2]/JDet;
  JI[2,2] := J[1,1]/JDet;
  JI[1,2] := -J[1,2]/JDet;
  JI[2,1] := -J[2,1]/JDet;

  //Calculate the cartesian derivatives
  for iC := 1 to 2 do
  begin
    for jC := 1 to 8 do
    begin
      CDeriv[iC,jC] := 0;
      for kC := 1 to 2 do
      begin
        CDeriv[iC,jC] := CDeriv[iC,jC] + JI[iC,kC]*SFDeriv[kC,jC];
      end;
    end;
  end;

end;

////////////////////////////////////////////////////////////////////////
// Calculates the shape functions and their cartesian derivatives //
////////////////////////////////////////////////////////////////////////

Procedure TElement.CalcShape( u,v : Double );
begin
  //Shape functions
  SFunc[1] := -1/4 * (1-u)*(1-v)*(1+u+v);
  SFunc[2] :=  1/2 * (1-u*u)*(1-v);
  SFunc[3] :=  1/4 * (1+u)*(1-v)*(u-v-1);
  SFunc[4] :=  1/2 * (1+u)*(1-v*v);
  SFunc[5] :=  1/4 * (1+u)*(1+v)*(u+v-1);
  SFunc[6] :=  1/2 * (1-u*u)*(1+v);
  SFunc[7] :=  1/4 * (1-u)*(1+v)*(-u+v-1);
  SFunc[8] :=  1/2 * (1-u)*(1-v*v);

  //Shape function derivatives
  SFDeriv[1,1] :=  1/4 * (v+2*u-2*u*v-v*v);
  SFDeriv[1,2] :=  -u+u*v;
  SFDeriv[1,3] :=  1/4 * (-v+2*u-2*u*v+v*v);
  SFDeriv[1,4] :=  1/2 * (1-v*v);
  SFDeriv[1,5] :=  1/4 * (v+2*u+2*u*v+v*v);
  SFDeriv[1,6] :=  -u-u*v;
  SFDeriv[1,7] :=  1/4 * (-v+2*u+2*u*v-v*v);
  SFDeriv[1,8] :=  1/2 * (-1+v*v);
  SFDeriv[2,1] :=  1/4 * (u+2*v-u*u-2*u*v);
  SFDeriv[2,2] :=  1/2 * (-1+u*u);
  SFDeriv[2,3] :=  1/4 * (-u+2*v-u*u+2*u*v);
  SFDeriv[2,4] :=  -v-u*v;
  SFDeriv[2,5] :=  1/4 * (u+2*v+u*u+2*u*v);
  SFDeriv[2,6] :=  1/2 * (1-u*u);
  SFDeriv[2,7] :=  1/4 * (-u+2*v+u*u-2*u*v);
  SFDeriv[2,8] :=  -v+u*v;
```

```
end;


////////////////////////////////////////////////
// Sets up the numerical integration constants //
////////////////////////////////////////////////

Procedure TElement.SetupNumInt;
begin

  case NumIntOrder of
    2 : begin
          GaussPos[1] := -0.577350269189626;
          GaussWgt[1] := 1;
          GaussPos[2] := 0.577350269189626;
          GaussWgt[2] := 1;
        end;

    3 : begin
          GaussPos[1] := -0.774596669241483;
          GaussPos[2] := 0;
          GaussWgt[1] := 0.555555555555556;
          GaussWgt[2] := 0.888888888888889;
          GaussPos[3] := 0.774596669241483;
          GaussPos[4] := 0;
          GaussWgt[4] := 0.555555555555556;
          GaussWgt[4] := 0.888888888888889;
        end;
  end;

end;

////////////////////////////////////////////////////
// Calculates the plate element stiffness matrix //
////////////////////////////////////////////////////

Procedure TElement.CalcStiffp;
var
  iCount, jCount, kCount : Integer;  //Loop counters
  lCount, mCount, nCount : Integer;  //Loop counters
  uPoint, vPoint         : Double;   //Sampling point indices
  Area, JDet             : Double;   //Differential area and Jacobian determinant

begin

  //Zero all matrix entries
  for iCount := 1 to 24 do
    for jCount := 1 to 24 do
      ElStiffp[iCount,jCount] := 0;

  //Calculate the D matrix
  SetupNumInt;
  CalcDp;
  kCount := 0;

  //Numerical integration
  for iCount := 1 to 2 do
  begin
    uPoint := GaussPos[iCount];
    for jCount := 1 to 2 do
    begin
      vPoint := GaussPos[jCount];
      inc(kCount);

      CalcShape(uPoint,vPoint);
      CalcJ(JDet,kCount);
      Area := JDet*GaussWgt[iCount]*GaussWgt[jCount];

      CalcBp;
      CalcDBp;
```

```
      for lCount := 1 to 24 do
      begin
        for mCount := lCount to 24 do
        begin
          for nCount := 1 to 5 do
          begin
            ElStiffp[lCount,mCount] := ElStiffp[lCount,mCount]
                                       + Bp[nCount,lCount] * DBp[nCount,mCount] * Area;
          end;
        end;
      end;

      for lCount := 1 to 5 do
      begin
        for mCount := 1 to 24 do
        begin
          GaussDBp[lCount,mCount,kCount] := DBp[lCount,mCount];
        end;
      end;

    end;
  end;

  for lCount := 1 to 24 do
  begin
    for mCount := 1 to 24 do
    begin
      ElStiffp[mCount,lCount] := ElStiffp[lCount,mCount];
    end;
  end;

end;

////////////////////////////////////////////////////
// Calculates the membrane element stiffness matrix //
////////////////////////////////////////////////////

Procedure TElement.CalcStiffm;
var
  iCount, jCount, kCount : Integer; //Loop counters
  lCount, mCount, nCount : Integer; //Loop counters
  uPoint, vPoint         : Double;  //Sampling point indices
  Area, JDet             : Double;  //Differential area and Jacobian determinant

begin

  //Zero all matrix entries
  for iCount := 1 to 16 do
    for jCount := 1 to 16 do
      ElStiffm[iCount,jCount] := 0;

  //Calculate the D matrix
  SetupNumInt;
  CalcDm;
  kCount := 0;

  //Numerical integration
  for iCount := 1 to 2 do
  begin
    uPoint := GaussPos[iCount];
    for jCount := 1 to 2 do
    begin
      vPoint := GaussPos[jCount];
      inc(kCount);

      CalcShape(uPoint,vPoint);
```

```
      CalcJ(JDet,kCount);
      Area := JDet*GaussWgt[iCount]*GaussWgt[jCount]*h;

      CalcBm;
      CalcDBm;

      for lCount := 1 to 16 do
      begin
        for mCount := lCount to 16 do
        begin
          for nCount := 1 to 3 do
          begin
            ElStiffm[lCount,mCount] := ElStiffm[lCount,mCount]
                                       + Bm[nCount,lCount] * DBm[nCount,mCount] * Area;
          end;
        end;
      end;

      for lCount := 1 to 3 do
      begin
        for mCount := 1 to 16 do
        begin
          GaussDBm[lCount,mCount,kCount] := DBm[lCount,mCount];
        end;
      end;

    end;
  end;

  for lCount := 1 to 16 do
  begin
    for mCount := 1 to 16 do
    begin
      ElStiffm[mCount,lCount] := ElStiffm[lCount,mCount];
    end;
  end;

end;

////////////////////////////////////////////////
// Assemble the total element stiffness matrix //
////////////////////////////////////////////////

Procedure TElement.CalcStiff;
var
  i, j, k, l, m : Integer;  //Loop counters

begin
  CalcStiffm;
  CalcStiffp;
  for i := 1 to 8 do
  begin
    for j := 1 to 8 do
    begin
      for k := 1 to 2 do
      begin
        for l := 1 to 2 do
        begin
          ElStiff[5*i-(k+2),5*j-(l+2)] := ElStiffm[i*k,j*l];
        end;
      end;
      for k := 1 to 3 do
      begin
        for l := 1 to 3 do
        begin
          ElStiff[5*i-(k-1),5*j-(l-1)] := ElStiffp[i*k,j*l];
        end;
      end;
    end;
  end;
```

```
end;



////////////////////////////////////////
// Reduce UDL to equivalent nodal loads //
////////////////////////////////////////

Procedure Telement.CalcUDLoad;
var
  iCount, jCount, kCount : Integer;  //Loop counters
  lCount, pos            : Integer;  //Loop counters
  u, v                   : Double;   //Gauss point coordinates
  DArea, JDet            : Double;   //Differential area and Jacobian determinant

begin
  //Zero all matrix entries
  for iCount := 1 to 24 do
    ElLoad[iCount] := 0;

  //Numerical integration
  kCount := 0;
  for iCount := 1 to 2 do
  begin
    u := GaussPos[iCount];
    for jCount := 1 to 2 do
    begin
      v := GaussPos[jCount];
      inc(kCount);

      CalcShape(u,v);
      CalcJ(JDet,kCount);
      DArea := JDet*GaussWgt[iCount]*GaussWgt[jCount];

      for lCount := 1 to 8 do
      begin
        pos := (lCount-1)*3+1;
        ElLoad[pos] := ElLoad[pos] + SFunc[lCount]*UDL*DArea;
      end;

    end;
  end;
end;

////////////////////////////////////////////////
// Calculate shrinkage loads given curvatures //
////////////////////////////////////////////////


Procedure Telement.CalcShrinkLoad(CurvX, CurvY, CurvXY, ShearX, ShearY : double);
var
  i, j, k, l, m, posi : Integer;                    //Loop counters
  u, v                : Double;                     //Gauss point coordinates
  JDet                : Double;                     //Jacobian determinant
  DArea               : Double;                     //Differential area
  iStrain             : array[1..5] of double;      //Initial strain matrix
  BpT                 : array[1..24,1..5] of double; //Transponent of [B]
  BpTD                : array[1..24,1..5] of double; //Product [B^T][D]

  //Calculate transponent of [B]
  Procedure CalcBpTransponent;
  var
    i, j : integer;  //Loop counters

  begin
    for i := 1 to 5 do
    begin
      for j := 1 to 24 do
```

```
      begin
        BpT[j,i] := Bp[i,j];
      end;
    end;
  end;


  //Calculate the product of [B^T] and [D]
  Procedure CalcBpTD;
  var
    i, j, k : integer;  //Loop counters

  begin
    for i := 1 to 24 do
      for j := 1 to 5 do
        BpTD[i,j] := 0;
    for i := 1 to 24 do
      for j := 1 to 5 do
        for k := 1 to 5 do
          BpTD[i,j] := BpTD[i,j] + BpT[i,k]*Dp[k,j];
  end;

begin
  //Assign the initial strain matrix and calculate D
  iStrain[1] := CurvX;
  iStrain[2] := CurvY;
  iStrain[3] := CurvXY;
  iStrain[4] := ShearX;
  iStrain[5] := ShearY;
  CalcDp;


  //Zero all matrix entries
  for i := 1 to 24 do
    ElLoad[i] := 0;

  //Numerical integration
  k := 0;
  for i := 1 to 2 do
  begin
    u := GaussPos[i];
    for j := 1 to 2 do
    begin
      v := GaussPos[j];
      inc(k);

      CalcShape(u,v);
      CalcJ(JDet,k);
      DArea := JDet*GaussWgt[i]*GaussWgt[j];
      CalcBp;
      CalcBpTransponent;
      CalcBpTD;

      for l := 1 to 24 do
        for m := 1 to 5 do
            ElLoad[l] := ElLoad[l]+BpTD[l,m]*iStrain[m]*DArea;

    end;
  end;
end;

////////////////////////////////////////////////////////
// Calculate the average moments for a crack analysis //
////////////////////////////////////////////////////////

Procedure TElement.CalcAvgs;
var
  iCount : Integer;
begin
```

```
  MxAvg := 0;
  MyAvg := 0;

  for iCount := 1 to 4 do
  begin
    MxAvg := MxAvg + abs(SRes[iCount,4]) + abs(SRes[iCount,6]);
    MyAvg := MyAvg + abs(SRes[iCount,5]) + abs(SRes[iCount,6]);
  end;

  MxAvg := MxAvg/4;
  MyAvg := MyAvg/4;

end;

////////////////////////////////////////////////////////
// Calculates the curvatures due to restrained shrinkage //
////////////////////////////////////////////////////////


Procedure TElement.CalcShrinkageCurvatures;
var
  xs, ys         : Double; //Eccentricity of steel with respect to the cracked NA
  x, y           : Double; //NA of the transformed section
  Ec             : Double; //Effective concrete modulus
  n              : Double; //Modular ratio
  CurveX, CurveY : Double; //Shrinkage curvatures
  Ix, Iy         : Double; //Moments of inertia of the transformed section
Begin

  Ec := E/(1+phi);
  n  := Es/Ec;

  x  := (0.5*pow(h,2) + n*Astx*dtx)/(h+n*Astx);
  Ix := pow(h,3)/12+h*pow(h/2-x,2)+n*Astx*pow(dtx-x,2);
  xs := dtx - x;
  CurveX := ecs*n*Astx*xs/Ix;

  y  := (0.5*pow(h,2) + n*Asty*dty)/(h+n*Asty);
  Iy := pow(h,3)/12+h*pow(h/2-y,2)+n*Astx*pow(dty-y,2);
  ys := dty - y;
  CurveY := ecs*n*Asty*ys/Iy;

  CalcShrinkLoad(CurveX,CurveY,0,0,0);

end;

////////////////////////////////////////////////////////
// Calculates curvatures due given moment resultants //
////////////////////////////////////////////////////////

Procedure TElement.CalcCurvatures;
var
  Mx, My, Mxy : Double;  //Moments
  a, b, c, d  : Double;  //Temporary storage variables
  i           : Integer; //Loop counter

begin

  Mx := 0;
  My := 0;
  Mxy := 0;

  a := Dp[1,1]; b := Dp[1,2];
  c := Dp[2,1]; d := Dp[2,2];

  for i := 1 to 4 do
  begin
    Mx  := Mx  + SRes[i,4];
    My  := My  + SRes[i,5];
    Mxy := Mxy + SRes[i,5];
  end;
```

```
  Mx  := Mx/4;
  My  := Mxy/4;
  Mxy := My/4;

  Cxy := Mxy/Dp[3,3];
  Cy  := (My-c/a*Mx)/(d-c*b/a);
  Cx  := (Mx-b*Cy)/a;

end;
////////////////////////////////////
// Initializes several variables //
////////////////////////////////////

Procedure TElement.InitInertia;
begin
  Igx := 1/12*pow(h,3);
  Igy := 1/12*pow(h,3);
  Iex := Igx;
  Iey := Igy;
  IeOldx := Igx;
  IeOldy := Igy;
  alphax := 1;
  alphay := 1;
end;

//////////////////////////////
// Calculates creep factors //
//////////////////////////////

Procedure TElement.CalcCreepFactors;
var
  Ibarx, Ibary, Ix, Iy, x, y, AeffX, AeffY, ycx, ycy, delyx, delyy : double;
  xga, yga, xca, yca, Icxa, Icya : double;
  na, n, Ecreep      : double;
  Igbarx, Icbarx, Igbary, Icbary : double;
  xcprime, ycprime, xcaprime, ycaprime : double;
begin

  ECreep := E/(1+phi);
  na := Es/Ecreep;
  n := Es/E;

  xg  := (t*t/2 + n*Astx*dtx  + n*Ascx*(t-dcx))/ (t + n*Astx  + n*Ascx);
  yg  := (t*t/2 + n*Asty*dty  + n*Ascy*(t-dcy))/ (t + n*Asty  + n*Ascy);
  xga := (t*t/2 + na*Astx*dtx + na*Ascx*(t-dcx))/(t + na*Astx + na*Ascx);
  yga := (t*t/2 + na*Asty*dty + na*Ascy*(t-dcy))/(t + na*Asty + na*Ascy);

  xcprime := dtx*( -n*(rox+rocx) + sqrt(pow(n*(rox+rocx),2) + 2*n*(rox+(t-dcx)/dtx*rocx)) );
  ycprime := dty*( -n*(roy+rocy) + sqrt(pow(n*(roy+rocy),2) + 2*n*(roy+(t-dcy)/dty*rocy)) );

  xcaprime := dtx*( -na*(rox+rocx) + sqrt(pow(na*(rox+rocx),2)
              + 2*na*(rox+(t-dcx)/dtx*rocx)) );
  ycaprime := dty*( -na*(roy+rocy) + sqrt(pow(na*(roy+rocy),2)
              + 2*na*(roy+(t-dcy)/dty*rocy)) );

  xc  := xg*alphax; if xc<xcprime then xc:=xcprime else if xc>xg then xc:=xg;
  yc  := yg*alphay; if yc<ycprime then yc:=ycprime else if yc>yg then yc:=yg;
  xca := xga*alphax; if xca<xcaprime then xca:=xcaprime else if xca>xga then xca:=xga;
  yca := yga*alphay; if yca<ycaprime then yca:=ycaprime else if yca>yga then yca:=yga;

  IgBarx := pow(t,3)/12+t*pow(t/2-xga,2)+na*Astx*pow(dtx-xga,2)+na*Ascx*pow((t-dtx)-xga,2);
  IcBarx := 1/3*pow(xca,3) + na*rocx*pow(xca-(t-dcx),2)*dtx + na*rox*pow(dtx-xca,2)*dtx;
  IgBary := pow(t,3)/12+t*pow(t/2-yga,2)+na*Asty*pow(dty-yga,2)+na*Ascy*pow((t-dcy)-yga,2);
  IcBary := 1/3*pow(yca,3) + na*rocy*pow(yca-(t-dcy),2)*dty + na*roy*pow(dty-yca,2)*dty;

  if ((not CrackedX) or (alphax>9.99)) then
  begin
    Aeffx := t;
    Ix := pow(t,3)/12+t*pow(t/2-xga,2);
```

```
    ycx := Aeffx/2; ycx := ycx-xga;
    delyx := xga-xg;
    IBarx := IgBarx;
  end
  else
  begin
    Aeffx := xc;
    Ix := pow(xc,3)/12+xc*(xca-xc/2)*(xca-xc/2);

    ycx := Aeffx/2; ycx := ycx-xca;
    delyx := xca-xc;
    IBarx := IcBarX;
  end;

  if ((not CrackedY) or (alphaY>9.99)) then
  begin
    Aeffy := t;
    Iy := pow(t,3)/12+t*pow(t/2-yga,2);

    ycy := Aeffy/2; ycy := ycy-yga;
    delyy := yga-yg;
    IBary := IgBary;
  end
  else
  begin
    Aeffy := yc;
    Iy := pow(yc,3)/12+yc*(yca-y/2)*(yca-y/2);

    ycy := Aeffy/2; ycy := ycy-yca;
    delyy := yca-yc;
    IBary := IcBary;
  end;

  kappaX := (Ix + Aeffx*ycx*delyx)/IBarx;
  kappaY := (Iy + Aeffy*ycy*delyy)/IBary;

  CalcForCreep := True;
end;

////////////////////////////////
// Calculates crack factors //
////////////////////////////////

Procedure TElement.CalcModFactors(CrackType:integer);
var
  alpha      : Double;  //Modular ratio
  fr         : Double;  //Modulus of rupture (Should be an element property)
  yx, yy     : Double;  //NA positions of the gross concrete section
  dtxc, dtyc : Double;  //Embedment depth of compression reinforcement (Should be an element
                          property)

begin

  //Initialize
  Crackedx := False;
  Crackedy := False;
  IeOldx := Iex;
  IeOldy := Iey;
  alphaOldx := alphax;
  alphaOldy := alphay;
  alpha := Es/E;

  //Calculate NA's and moments of inertia
  xc := dtx*( -alpha*(rox+rocx) + sqrt(pow(alpha*(rox+rocx),2)
        + 2*alpha*(rox+(t-dcx)/dtx*rocx)) );
  Icx := 1/3*pow(xc,3)/12 + alpha*rocx*pow(xc-(t-dcx),2)*dtx + alpha*rox*pow(dtx-xc,2)*dtx;

  yc := dty*( -alpha*(roy+rocy) + sqrt(pow(alpha*(roy+rocy),2)
        + 2*alpha*(roy+(t-dcy)/dty*rocy)) );
  Icy := 1/3*pow(yc,3)/12 + alpha*rocy*pow(yc-(t-dcy),2)*dty + alpha*roy*pow(dty-yc,2)*dty;
```

```
  Igx := 1/12*pow(h,3);
  yx := h/2;

  Igy := 1/12*pow(h,3);
  yy := h/2;

  //Calculate cracking moments
  Mcrx := fr*Igx/yx;
  Mcry := fr*Igy/yy;

  //Bilinear method cracking factors
  if(MxAvg >= Mcrx) then ksix := 1-pow(Mcrx/MxAvg,2) else ksix := 0;
  if(MyAvg >= Mcry) then ksiy := 1-pow(Mcry/MyAvg,2) else ksiy := 0;

  if(MxAvg >= Mcrx) then
  begin
    if CrackType = 0 then
    begin
      Iex := pow(Mcrx/MxAvg,4)*Igx + (1 - pow(Mcrx/MxAvg,4))*Icx;
      Iex := IeOldx-(IeOldx - Iex)/2;
    end
    else
    begin
      Iex := (Igx*Icx) / ( (1-ksix)*Icx+ksix*Igx );
      Iex := IeOldx-(IeOldx - Iex)/2;
    end;
    if Iex > Igx then Iex := Igx
    else if Iex < Icx then Iex := Icx;
    alphax := Iex/Igx;
    CrackedX := True;
  end
  else
  begin
    Iex := Igx;
    IeOldx := Iex;
    alphax := 1;
    CrackedX := False;
  end;

  if(MyAvg >= Mcry) then
  begin
    if (CrackType = 0) then
    begin
      Iey := pow(Mcry/MyAvg,4)*Igy + (1 - pow(Mcry/MyAvg,4))*Icy;
      Iey := IeOldy-(IeOldy - Iey)/2;
    end
    else
    begin
      Iey := (Igy*Icy) / ( (1-ksiy)*Icy+ksiy*Igy );
      Iey := IeOldy-(IeOldy - Iey)/2;
    end;
    if Iey > Igy then Iey := Igy
    else if Iey < Icy then Iey := Icy;
    alphay := Iey/Igy;
    CrackedY := True;
  end
  else
  begin
    Iey := Igy;
    alphay := 1;
    CrackedY := False;
  end;

  if alphax>0.99 then crackedx := false;
  if alphay>0.99 then crackedy := false;

end;

end.
```

7.3 Elastic Analysis Procedure

```
procedure ElasticAnalysis;
var
  iCount: Integer;  //Loop counters

begin

  //Obtain FEA geometry, loading & material input
  readInput;

  //Initialize the model and global arrays
  Initialize;

  //Assemble the element and global stiffness matrices
  Assemble;

  //Assemble the load vector
  LoadVector;

  //Solve the linear system
  Solve;

  //Update global deflection array
  UpdateGlobalDVec;

  //Update each element's deflection array
  UpdateElementDVec;

  //Calculate and smooth stress resultants
  CalcResultants(true);

  UpdateOuput;

end;
```

7.4 Crack Analysis Procedure

```
procedure CrackAnalysis;
var
  iCount  : Integer;  //Loop counter
  El      : TElement; //Element object pointer
  Conv    : Double;   //Convergence variable
  limit   : Double;   //Convergence limit

  //Returns current convergence index
  function testConv:double;
  var
    i, j : integer;
    x,y : double;
  begin
    x := 0;
    y := 0;
    for i := 1 to num*num do
    begin
      x := x + abs(slab[i].alphax-slab[i].alphaOldx)/slab[i].alphaoldx;
      y := y + abs(slab[i].alphay-slab[i].alphaOldy)/slab[i].alphaoldy;
    end;
    x := x/(num*num);
    y := y/(num*num);
    result := max(x,y);
  end;

begin

  //Obtain FEA geometry, loading & material input
  readInput;

  //Initialize the model and global arrays
  Initialize;

  limit := 1e-3;  //Convergence limit

  //Initialize element moments of inertia
  for iCount := 1 to numElements do Slab[iCount].InitInertia;

  conv := 1;  //Convergence tester
  while ( abs(conv)>limit ) do
  begin
    //Record current deflection field
    StoreOrigDVec;

    //Assemble element and global stiffness matrices
    Assemble;

    //Assemble the load vector
    LoadVector;

    //Solve the linear system
    Solve;

    //Update global deflection array
    UpdateGlobalDVec;

    //Update elements' deflection array
    UpdateElementDVec;

    //Calculate stress resultants but do not smooth
    CalcResultants(false);

    //Calculate average moments and modification factors for each element
    for jCount := 1 to numElements do
    begin
      Slab[jCount].CalcAvgs;
      Slab[jCount].CalcModFactors(CrackType);
    end;
```

```
    //Update the convergence variable
    conv := testConv;

    if(conv < limit) then
      //If solution converged, smooth stress resultants
      CalcResultants(true);
    else
      //If solution not converged, restore original deflection field
      RegressGlobalDVec;

  end;

  UpdateOutput;
end;
```

7.5 Creep Analysis Procedure

```
Procedure CreepClickAnalysis;
var
  i  : Integer;  //Loop counter;

begin

  //Perform a crack analysis
  CrackAnalysis;

  //Initialize global arrays
  Initialize;

  //Calculate creep factors for each element
  for i := 1 to numElements do
  begin
    Slab[i].CalcCreepFactors;
  end;

  //Assemble element and global stiffness matrices
  Assemble;

  //Assemble the load vector
  LoadVector;

  //Solve the linear system
  Solve;

  //Update global deflection array
  UpdateGlobalDVec;

  //Update elements' deflection array
  UpdateElementDVec;

  //Calculate and smooth stress resultants
  CalcResultants(true);

  UpdateOutput;

end;
```

7.6 Shrinkage Analysis Procedure

```
Procedure ShrinkAnalysis;
var
  i : integer;

begin
  //Obtain FEA geometry, loading & material input
  readInput;

  //Initialize the model and global arrays
  Initialize;

  //Assemble element and global stiffness matrices
  Assemble;

  //Calculate shrinkage forces for each element
  for i := 1 to num*num do
  begin
    slab[i].ecs := ecs;
    slab[i].CalcShrinkageMoments;
  end;

  //Assemble the load vector
  LoadVector;

  //Solve the linear system
  Solve;

  //Update global deflection array
  UpdateGlobalDVec;

  //Update elements' deflection array
  UpdateElementDVec;

  //Calculate and smooth stress resultants
  CalcResultants(true);

  UpdateOutput;

end;
```

7.7 Stiffness Matrix Assembly Procedure

```
Procedure Assemble;
var
  iCount, jCount, kCount : Integer;  //Loop counters
  GNodeRow, GNodeCol     : Integer;  //Row, column indices
  DOF                    : Integer;  //Degrees of freedom variables
begin

  DOF := 5;

  for iCount := 1 to numElements do
  begin
    //Calculate element stiffness matrices
    Slab[iCount].CalcStiff;
    for jCount := 1 to 8 do
    begin

      GNodeRow := Slab[iCount].NodesNum[jCount];
      for kCount := 1 to 8 do
      begin
        //Place elemnt matrix in structure stiffness matrix
        GNodeCol := Slab[iCount].NodesNum[kCount];
        GStiff[5*GNodeRow-4,5*GNodeCol-4] := GStiff[5*GNodeRow-4,5*GNodeCol-4]
                                             + Slab[iCount].ElStiffm[2*jCount-1,2*kCount-1];
        GStiff[5*GNodeRow-4,5*GNodeCol-3] := GStiff[5*GNodeRow-4,5*GNodeCol-3]
                                             + Slab[iCount].ElStiffm[2*jCount-1,2*kCount-0];

        GStiff[5*GNodeRow-3,5*GNodeCol-4] := GStiff[5*GNodeRow-3,5*GNodeCol-4]
                                             + Slab[iCount].ElStiffm[2*jCount-0,2*kCount-1];
        GStiff[5*GNodeRow-3,5*GNodeCol-3] := GStiff[5*GNodeRow-3,5*GNodeCol-3]
                                             + Slab[iCount].ElStiffm[2*jCount-0,2*kCount-0];

        GStiff[3*GNodeRow-2,3*GNodeCol-2] := GStiff[3*GNodeRow-2,3*GNodeCol-2]
                                             + Slab[iCount].ElStiffp[3*jCount-2,3*kCount-2];
        GStiff[3*GNodeRow-2,3*GNodeCol-1] := GStiff[3*GNodeRow-2,3*GNodeCol-1]
                                             + Slab[iCount].ElStiffp[3*jCount-2,3*kCount-1];
        GStiff[3*GNodeRow-2,3*GNodeCol-0] := GStiff[3*GNodeRow-2,3*GNodeCol-0]
                                             + Slab[iCount].ElStiffp[3*jCount-2,3*kCount-0];

        GStiff[3*GNodeRow-1,3*GNodeCol-2] := GStiff[3*GNodeRow-1,3*GNodeCol-2]
                                             + Slab[iCount].ElStiffp[3*jCount-1,3*kCount-2];
        GStiff[3*GNodeRow-1,3*GNodeCol-1] := GStiff[3*GNodeRow-1,3*GNodeCol-1]
                                             + Slab[iCount].ElStiffp[3*jCount-1,3*kCount-1];
        GStiff[3*GNodeRow-1,3*GNodeCol-0] := GStiff[3*GNodeRow-1,3*GNodeCol-0]
                                             + Slab[iCount].ElStiffp[3*jCount-1,3*kCount-0];

        GStiff[3*GNodeRow-0,3*GNodeCol-2] := GStiff[3*GNodeRow-0,3*GNodeCol-2]
                                             + Slab[iCount].ElStiffp[3*jCount-0,3*kCount-2];
        GStiff[3*GNodeRow-0,3*GNodeCol-1] := GStiff[3*GNodeRow-0,3*GNodeCol-1]
                                             + Slab[iCount].ElStiffp[3*jCount-0,3*kCount-1];
        GStiff[3*GNodeRow-0,3*GNodeCol-0] := GStiff[3*GNodeRow-0,3*GNodeCol-0]
                                             + Slab[iCount].ElStiffp[3*jCount-0,3*kCount-0];
      end;
    end;
  end;

end;
```

7.8 Gauss Reduction Procedure

```
Procedure Solve;
var
  Pivot, Factor, Residual : Double;
  i, j, k, l              : Integer;  //Loop counters
  iRow, iCol              : Integer;  //Row & column indices
  NBack, NBac1, Neqns1    : Integer;
  DOF, SR                 : Integer;  //Degrees of freedom and no of stress resultants
  Diff : Double;
begin
  DOF := 5;
  SR := 8;

  //Gauss Reduction
  for i := 1 to numNodes*DOF do
  begin
    if (SVec[i]<>1) then //free DOF
    begin
      Pivot := GStiff[i,i];
      if(abs(pivot)>1e-10) then
      begin //valid pivot
        if (i<numNodes*DOF) then
        begin
          j := i+1;
          for iRow := j to numNodes*DOF do
          begin
            Factor := GStiff[iRow,i]/Pivot;
            if (Factor<>0) then
            begin
              for iCol := i to numNodes*DOF do
              begin
                GStiff[iRow,iCol] := GStiff[iRow,iCol] - Factor*GStiff[i,iCol];
              end;
              LVec[iRow] := LVec[iRow] - Factor*LVec[i];
            end;
          end;
        end;
      end
      else //invalid pivot
      begin
        infomsg('Non positive definite matrix');
        exit;
      end;
    end
    else //restrained DOF
    begin
      for iRow := i to numNodes*DOF do
      begin
        LVec[iRow] := LVec[iRow] - GStiff[iRow,i]*FVec[i];
        GStiff[iRow,i] := 0;
      end;
    end;
  end;

  //Back substitution process
  j := numNodes*DOF+1;
  for i := 1 to numNodes*DOF do
  begin
    k := j-i; //nback
    Pivot := GStiff[k,k];
    Residual := LVec[k];

    if (k<>numNodes*DOF) then
    begin
      l := k+1;
      for iCol := l to numNodes*DOF do
      begin
        Residual := Residual - GStiff[k,iCol]*DVec[iCol];
      end;
```

```
    end;

    If (SVec[k]=0) then
    begin
      DVec[k] := Residual/Pivot;
    end
    else
    begin
      DVec[k] := FVec[k];
      RVec[k] := -Residual;
    end;

  end;

end;
```

7.9 Stress Resultant Calculation Procedure

```
Procedure CalcResultants(smooth:boolean);
var
  iCount, jCount, kCount, lCount : Integer;                    //Loop counters
  mCount, nCount, oCount, pCount : Integer;                    //Loop counters
  trans                          : array[1..4,1..4] of Double; //Extrapolation matrix
  temp                           : array[1..8] of Double;      //Holder matrix
  Sum                            : array[1..8] of Double;      //Holder matrix
  SR : Integer;
  NodesArray : array[1..4] of integer;

begin
  SR := 8;

  //Calculate stress resultants at sampling points - membrane
  for iCount := 1 to numElements do
  begin

    lCount := 0; //1 to 4
    for jCount := 1 to 2 do
    begin
      for kCount := 1 to 2 do
      begin
        inc(lCount);
        for mCount := 1 to 3 do
        begin
          nCount := 0;
          Slab[iCount].SRes[lCount,mCount] := 0;

          for oCount := 1 to 8 do
          begin
            for pCount := 1 to 2 do
            begin
              inc(nCount);
              Slab[iCount].SRes[lCount,mCount] := Slab[iCount].SRes[lCount,mCount] +
                (Slab[iCount].GaussDBm[mCount,nCount,lCount] *
                  Slab[iCount].Def[pCount,oCount]);
            end;
          end;
        end;
      end;
    end;

  end;

  //Calculate stress resultants at sampling points - plate
  for iCount := 1 to numElements do
  begin

    lCount := 0; //1 to 4
    for jCount := 1 to 2 do
    begin
      for kCount := 1 to 2 do
      begin
        inc(lCount);
        for mCount := 4 to 8 do
        begin
          nCount := 0;
          Slab[iCount].SRes[lCount,mCount] := 0;

          for oCount := 1 to 8 do
          begin
            for pCount := 3 to 5 do
            begin
              inc(nCount);
              Slab[iCount].SRes[lCount,mCount] := Slab[iCount].SRes[lCount,mCount] +
                (Slab[iCount].GaussDBp[mCount-3,nCount,lCount] *
                  Slab[iCount].Def[pCount,oCount]);
            end;
```

```
        end;
      end;
    end;
  end;
  Slab[iCount].CalcCurvatures;
end;

if not smooth then exit;

//Smooth by bi-linear extrapolation to the corner nodes
trans[1,1]:=1.866025404;
trans[1,2]:=-0.5;
trans[1,3]:=0.133974596;
trans[1,4]:=-0.5;
trans[2,1]:=-0.5;
trans[2,2]:=1.866025404;
trans[2,3]:=-0.5;
trans[2,4]:=0.133974596;
trans[3,1]:=0.133974596;
trans[3,2]:=-0.5;
trans[3,3]:=1.866025404;
trans[3,4]:=-0.5;
trans[4,1]:=-0.5;
trans[4,2]:=0.133974596;
trans[4,3]:=-0.5;
trans[4,4]:=1.866025404;

for iCount := 1 to numElements do
begin
  for jCount := 1 to SR do
      temp[jCount] := Slab[iCount].SRes[2,jCount];
  for kCount := 1 to SR do
      Slab[iCount].SRes[2,kCount] := Slab[iCount].SRes[3,kCount];
  for kCount := 1 to SR do
      Slab[iCount].SRes[3,kCount] := Slab[iCount].SRes[4,kCount];
  for kCount := 1 to SR do
      Slab[iCount].SRes[4,kCount] := temp[kCount];
end;

for iCount := 1 to num*num do
begin
  for jCount := 1 to SR do
  begin
    Slab[iCount].SResM[1,jCount] :=
      Slab[iCount].SRes[1,jCount]*trans[1,1] +
        Slab[iCount].SRes[2,jCount]*trans[1,2] +
          Slab[iCount].SRes[3,jCount]*trans[1,3] +
            Slab[iCount].SRes[4,jCount]*trans[1,4];
    Slab[iCount].SResM[3,jCount] :=
      Slab[iCount].SRes[1,jCount]*trans[2,1] +
        Slab[iCount].SRes[2,jCount]*trans[2,2] +
          Slab[iCount].SRes[3,jCount]*trans[2,3] +
            Slab[iCount].SRes[4,jCount]*trans[2,4];
    Slab[iCount].SResM[2,jCount] := (Slab[iCount].SResM[1,jCount]
                                     + Slab[iCount].SResM[3,jCount])/2;

    Slab[iCount].SResM[5,jCount] :=
      Slab[iCount].SRes[1,jCount]*trans[3,1] +
        Slab[iCount].SRes[2,jCount]*trans[3,2] +
          Slab[iCount].SRes[3,jCount]*trans[3,3] +
            Slab[iCount].SRes[4,jCount]*trans[3,4];
    Slab[iCount].SResM[4,jCount] := (Slab[iCount].SResM[3,jCount]
                                     + Slab[iCount].SResM[5,jCount])/2;

    Slab[iCount].SResM[7,jCount] :=
      Slab[iCount].SRes[1,jCount]*trans[4,1] +
        Slab[iCount].SRes[2,jCount]*trans[4,2] +
          Slab[iCount].SRes[3,jCount]*trans[4,3] +
            Slab[iCount].SRes[4,jCount]*trans[4,4];
    Slab[iCount].SResM[6,jCount] := (Slab[iCount].SResM[5,jCount]
                                     + Slab[iCount].SResM[7,jCount])/2;
```

```
      Slab[iCount].SResM[8,jCount] := (Slab[iCount].SResM[1,jCount]
                                       + Slab[iCount].SResM[7,jCount])/2;
    end;
  end;

  //Average stresses at nodal points to obtain unique values
  for iCount := 1 to numNodes do
  begin
    kCount := 0;
    for jCount := 1 to SR do
      Sum[jCount]:=0;
    for jCount := 1 to num*num do
    begin
      lCount := scanl(iCount,Slab[jCount].NodesNum,8);
      if lCount<>0 then
      begin
        inc(kCount);
        for mCount := 1 to SR do
          Sum[mCount] := Sum[mCount] + Slab[jCount].SResM[lCount,mCount];
      end;
    end;
    if kCount >0 then
    begin
      for jCount := 1 to SR do
        SRes[iCount,jCount] := Sum[jCount]/kCount;
    end;
  end;

  jCount := 1;
  for iCount := 1 to num*num do
  begin
    for kCount := 1 to 8 do
    begin
      if pos = Slab[iCount].NodesNum[kCount] then
      begin
        NodesArray[jCount] := iCount;
        jCount := jCount+1;
      end;
    end;
  end;

end;
```