

Chapter 13

SUPER-CONVERGENT PATCH RECOVERY

13.1 Patch Implementation Database

Since the super-convergent patch (SCP) recovery method is relatively easy to understand and is accurate for a wide range of problems, it was selected for implementation in the educational program MODEL. Its implementation is designed for use with most of the numerically integrated 1-D, 2-D, and 3-D elements in the MODEL library. Most of the literature on the SCP recovery methods are limited to a single element type and a single patch type. The present version is somewhat more general in allowing a mixture of element shapes in the mesh and a mesh that is either linear, quadratic, or cubic in its polynomial degree.

This represents actually the third version of the SCP algorithm and is a simplification of the first two. The method was originally developed for a p -adaptive code where all the elements could have a different polynomial degree on each element edge. That version was then extended to an object-oriented F90 p -adaptive program that also included equilibrium error contributions as suggested by Wiberg [10] and others. Including the p -adaptive features and the object-oriented features made the data base more complicated and required more planning and programming than desirable in an introductory text such as this one. However, the version given here has shown to be robust and useful.

The SCP recovery process is clearly heuristic in nature, so some arbitrary choices need to be made in the implementation. We begin by defining a "patch" to be a local group of elements surrounding at least one interior node or being adjacent to a boundary node. The original research in SCP recovery methods used patches sequentially built around each node in the mesh. Later it was widely recognized that one could use a patch for every element in the mesh. Therefore, three types of patches will be defined here:

1. Node-based patch: An adjacent group of elements associated with a particular node.
2. Element-based patch: All elements adjacent to a particular element.
3. Face-based patch: This subset of the element-based patch includes only the adjacent elements that share a common face with the selected element. For two-dimensional

elements this means that they share a common edge.

Those three choices for patches were shown in Fig. 12.2.2. Any of these three definitions of a patch requires that one have a mesh "neighbors list". That is, we will need a list of elements adjacent to each node, or a list of elements adjacent to each element, or the subset list of facing element neighbors. These can be expensive lists to create, but are often needed for other purposes and are sometime supplied by a mesh generation code or an equation re-ordering program.

Here several routines are included for creating the lists and printing them. Normally, since those neighbor lists are of unknown variable lengths, they would be stored in a linked list data structure. Here, for simplicity, they have been placed in rectangular arrays and padded with trailing zeros. This wastes a little storage space but keeps the general code simpler. Several of these routines are actually invoked at the mesh input stage as part of the data checking process. The neighbor lists are usually quite large and are not usually listed but can be (via keywords *list_el_to_el* or *pt_el_list*).

The primary routines for the node-based patches are the subroutines COUNT_L_AT_NODE and FORM_L_ADJACENT_NODE, seen in Figs.13.1.1 and 2, while the

```

SUBROUTINE COUNT_L_AT_NODES (N_ELEMS, NOD_PER_EL, MAX_NP, & ! 1
                           NODES, L_TO_N_SUM) ! 2
! * * * * * ! 3
! COUNT NUMBER OF ELEMENTS ADJACENT TO EACH NODE ! 4
! (TO SIZE ELEM ADJACENT TO ELEM LIST) ! 5
! * * * * * ! 6
IMPLICIT NONE ! 7
INTEGER, INTENT(IN) :: N_ELEMS, NOD_PER_EL, MAX_NP ! 8
INTEGER, INTENT(IN) :: NODES (N_ELEMS, NOD_PER_EL) ! 9
INTEGER, INTENT(OUT) :: L_TO_N_SUM (MAX_NP) !10
!11
INTEGER :: ELEM_NODES (NOD_PER_EL) !12
INTEGER :: IE, IN, N_TEMP !13
!14
! ELEM_NODES = INCIDENCES ARRAY FOR A SINGLE ELEMENT !15
! L_TO_N_SUM (I) = NUMBER OF ELEM NEIGHBORS OF NODE I !16
! MAX_NP = TOTAL NUMBER OF NODES !17
! N_ELEMS = TOTAL NUMBER OF ELEMENTS !18
! NOD_PER_EL = NUMBER OF NODES PER ELEMENT !19
! NODES = SYSTEM ARRAY OF ALL ELEMENTS INCIDENCES !20
!21
L_TO_N_SUM = 0 ! INITIALIZE !22
DO IE = 1, N_ELEMS ! LOOP OVER ALL ELEMENTS !23
! EXTRACT INCIDENCES LIST FOR ELEMENT IE !24
CALL ELEMENT_NODES (IE, NOD_PER_EL, NODES, ELEM_NODES) !25
DO IN = 1, NOD_PER_EL ! loop over each node !26
N_TEMP = ELEM_NODES (IN) !27
IF ( N_TEMP < 1 ) CYCLE ! to a real node !28
L_TO_N_SUM (N_TEMP) = L_TO_N_SUM (N_TEMP) + 1 !29
END DO ! over IN !30
END DO ! over elements !31
END SUBROUTINE COUNT_L_AT_NODES !32

```

Figure 13.1.1 Computing the neighbor array sizes

```

SUBROUTINE FORM_L_ADJACENT_NODES (N_ELEMS, NOD_PER_EL, MAX_NP, & ! 1
                                NODES, NEIGH_N, L_TO_N_NEIGH) ! 2
! * * * * * ! 3
!          TABULATE ELEMENTS ADJACENT TO EACH NODE ! 4
!          (TO SIZE ELEM ADJACENT TO ELEM LIST) ! 5
! * * * * * ! 6
IMPLICIT NONE ! 7
INTEGER, INTENT(IN) :: N_ELEMS, NOD_PER_EL, MAX_NP, NEIGH_N ! 8
INTEGER, INTENT(IN) :: NODES (N_ELEMS, NOD_PER_EL) ! 9
INTEGER, INTENT(OUT) :: L_TO_N_NEIGH (NEIGH_N, MAX_NP) !10
!11
INTEGER :: ELEM_NODES (NOD_PER_EL), COUNT (MAX_NP) ! scratch !12
INTEGER :: IE, IN, N_TEMP !13
!14
! ELEM_NODES = INCIDENCES ARRAY FOR A SINGLE ELEMENT !15
! L_TO_N_SUM (I) = NUMBER OF ELEM NEIGHBORS OF NODE I !16
! MAX_NP = TOTAL NUMBER OF NODES !17
! N_ELEMS = TOTAL NUMBER OF ELEMENTS !18
! NOD_PER_EL = NUMBER OF NODES PER ELEMENT !19
! NODES = SYSTEM ARRAY OF INCIDENCES OF ALL ELEMENTS !20
!21
L_TO_N_NEIGH = 0 ; COUNT = 0 ! INITIALIZE !22
DO IE = 1, N_ELEMS ! LOOP OVER ALL ELEMENTS !23
!24
!          EXTRACT INCIDENCES LIST FOR ELEMENT IE !25
CALL ELEMENT_NODES (IE, NOD_PER_EL, NODES, ELEM_NODES) !26
!27
DO IN = 1, NOD_PER_EL ! loop over each node !28
N_TEMP = ELEM_NODES (IN) !29
IF ( N_TEMP < 1 ) CYCLE ! to a real node !30
COUNT (N_TEMP) = COUNT (N_TEMP) + 1 !31
L_TO_N_NEIGH (COUNT (N_TEMP), N_TEMP) = IE !32
END DO ! over IN nodes !33
END DO ! over elements !34
END SUBROUTINE FORM_L_ADJACENT_NOD !35

```

Figure 13.1.2 Find elements at every node

element-based patches use the two similar subroutines COUNT_ELEMS_AT_ELEM and FORM_ELEMS_AT_EL which are given in Figs. 13.1.3 and 4. These routines are also useful in validating meshes that have been prepared by hand. Building lists of neighbors can take a lot of processing but they useful in plotting and post-processing.

In MODEL the default is to use an element-based patch. However, one can investigate other options by utilizing some of the available control keywords given in Fig. 13.1.5. Having selected a patch type, we should now give consideration to the kind of data that will be needed for the SCP recovery. There are two main segments in the process:

1. Averaging the patch and system nodal fluxes.
2. Using the system nodal fluxes in the calculation of an error estimate.

The whole basis of the SCP recovery is that there are special locations within an element where we can show that the derivatives are most accurate or exact for a given polynomial degree. We refer to such locations as element super-convergent points. They

```

SUBROUTINE COUNT_ELEMS_AT_ELEM (N_ELEMS, NOD_PER_EL, MAX_NP, & ! 1
      L_FIRST, L_LAST, NODES, NEEDS, L_TO_L_SUM, N_WARN) ! 2
! * * * * * ! 3
! COUNT NUMBER OF ELEMENTS ADJACENT TO OTHER ELEMENTS ! 4
! * * * * * ! 5
IMPLICIT NONE ! 6
INTEGER, INTENT(IN)      :: N_ELEMS, NOD_PER_EL, MAX_NP, NEEDS ! 7
INTEGER, INTENT(IN)      :: L_FIRST (MAX_NP), L_LAST (MAX_NP) ! 8
INTEGER, INTENT(IN)      :: NODES (N_ELEMS, NOD_PER_EL) ! 9
INTEGER, INTENT(OUT)     :: L_TO_L_SUM (N_ELEMS) !10
INTEGER, INTENT(INOUT)   :: N_WARN !11
!12
INTEGER :: ELEM_NODES (NOD_PER_EL), NEIG_NODES (NOD_PER_EL) !13
INTEGER :: FOUND, IE, IN, L_TEST, L_START, L_STOP, N_TEST !14
INTEGER :: NEED, KOUNT, NULLS !15
!16
! ELEM_NODES      = INCIDENCES ARRAY FOR A SINGLE ELEMENT !17
! KOUNT           = NUMBER OF COMMON NODES !18
! L_FIRST (I)     = ELEMENT WHERE NODE I FIRST APPEARS !19
! L_LAST (I)      = ELEMENT WHERE NODE I LAST APPEARS !20
! L_TO_L_SUM (I)  = NUMBER OF ELEM NEIGHBORS OF ELEMENT I !21
! MAX_NP         = TOTAL NUMBER OF NODES !22
! NEEDS          = NUMBER OF COMMON NODES TO BE A NEIGHBOR !23
! N_ELEMS        = TOTAL NUMBER OF ELEMENTS !24
! NOD_PER_EL     = NUMBER OF NODES PER ELEMENT !25
! NODES          = SYSTEM ARRAY OF INCIDENCES OF ALL ELEMENTS !26
!27
L_TO_L_SUM = 0 ; NEED = MAX (1, NEEDS) ! INITIALIZE !28
!29
MAIN : DO IE = 1, N_ELEMS ! LOOP OVER ALL ELEMENTS !30
      FOUND = 0 ! INITIALIZE !31
!32
! EXTRACT INCIDENCES LIST FOR ELEMENT IE !33
      CALL ELEMENT_NODES (IE, NOD_PER_EL, NODES, ELEM_NODES) !34
!35
! ESTABLISH RANGE OF POSSIBLE ELEMENT NEIGHBORS !36
      L_START = N_ELEMS ; L_STOP = 0 !37
      DO IN = 1, NOD_PER_EL !38
        N_TEST = ELEM_NODES (IN) !39
        IF ( N_TEST < 1 ) CYCLE! to a real node !40
        L_START = MIN (L_START, L_FIRST (N_TEST) ) !41
        L_STOP = MAX (L_STOP, L_LAST (N_TEST) ) !42
      END DO !43
!44

```

Figure 13.1.3a Interface and data for elements joining element

```

!      LOOP OVER POSSIBLE ELEMENT NEIGHBORS                                !45
IF ( L_START <= L_STOP ) THEN                                           !46
  RANGE : DO L_TEST = L_START, L_STOP                                     !47
    IF ( L_TEST /= IE ) THEN                                           !48
      KOUNT = 0 ! NO COMMON NODES                                       !49
                                                                    !50
!      LOOP OVER INCIDENCES OF POSSIBLE ELEMENT NEIGHBOR                !51
CALL ELEMENT_NODES (L_TEST,NOD_PER_EL,NODES,NEIG_NODES)                 !52
LOCAL : DO IN = 1, NOD_PER_EL                                           !53
  N_TEST = NEIG_NODES (IN)                                             !54
  IF ( N_TEST < 1 .OR. N_TEST > MAX_NP ) THEN                          !55
    PRINT *, 'INVALID NODE ', N_TEST, ' AT ', L_TEST                   !56
    N_WARN = N_WARN + 1 ! INCREMENT WARNING                             !57
    CYCLE LOCAL ! to a real node                                        !58
  END IF ! IMPOSSIBLE NODE                                             !59
  IF ( L_FIRST (N_TEST) > IE ) CYCLE LOCAL ! to next node              !60
  IF ( L_LAST  (N_TEST) < IE ) CYCLE LOCAL ! to next node              !61
                                                                    !62
!      COMPARE WITH INCIDENCES OF ELEMENT IE                             !63
IF ( ANY ( ELEM_NODES == N_TEST ) ) THEN                                 !64
  KOUNT = KOUNT + 1                                                    !65
  IF ( KOUNT == NEED ) THEN ! IS A NEIGHBOR                             !66
    FOUND = FOUND + 1                                                  !67
    EXIT LOCAL ! this L_TEST element search loop                       !68
  END IF ! NUMBER NEEDED                                              !69
  END IF                                                                !70
  END DO LOCAL ! over in                                              !71
  END IF                                                                !72
  END DO RANGE ! over candidate element L_TEST                         !73
  END IF ! a possible candidate                                       !74
  L_TO_L_SUM (IE) = FOUND                                             !75
END DO MAIN ! over all elements                                        !76
                                                                    !77
PRINT *, 'MAXIMUM NUMBER OF ELEMENT NEIGHBORS = ', &                   !78
        MAXVAL (L_TO_L_SUM)                                           !79
NULLS = COUNT ( L_TO_L_SUM == 0 ) ! CHECK DATA                       !80
IF ( NULLS > 0 ) THEN                                                 !81
  PRINT *, 'WARNING, ', NULLS, ' ELEMENTS HAVE NO NEIGHBORS'         !82
  N_WARN = N_WARN + 1 ! INCREMENT WARNING                             !83
  END IF                                                                !84
END SUBROUTINE COUNT_ELEMS_AT_ELEM                                     !85

```

Figure 13.1.3b Fill the neighbor array and validate


```

!      LOOP OVER POSSIBLE ELEMENT NEIGHBORS                ! 56
IF ( L_START <= L_STOP ) THEN                             ! 57
  RANGE : DO L_TEST = L_START, L_STOP                     ! 58
    KOUNT = 0 ! NO COMMON NODES                           ! 59
                                                    ! 60
!      EXTRACT NODES OF L_TEST                             ! 61
  CALL ELEMENT_NODES (L_TEST,NOD_PER_EL,NODES,NEIG_NODES) ! 62
                                                    ! 63
!      LOOP OVER INCIDENCES OF POSSIBLE ELEMENT NEIGHBOR ! 64
  LOCAL : DO IN = 1, NOD_PER_EL                           ! 65
    N_TEST = NEIG_NODES (IN)                              ! 66
    IF ( N_TEST < 1 ) CYCLE ! to a real node               ! 67
    IF (L_FIRST (N_TEST) > IE) CYCLE LOCAL ! to next node ! 68
    IF (L_LAST  (N_TEST) < IE) CYCLE LOCAL ! to next node ! 69
                                                    ! 70
!      COMPARE WITH INCIDENCES OF ELEMENT IE              ! 71
  IF ( ANY ( ELEM_NODES == N_TEST ) ) THEN                ! 72
    KOUNT = KOUNT + 1 ! SHARED NODE COUNT                 ! 73
    IF ( KOUNT == NEED ) THEN ! NEIGHBOR PAIR FOUND       ! 74
      FOUND = FOUND + 1 ! INSERT THE PAIR                 ! 75
                                                    ! 76
!      NOTE: THIS INSERT IS NOT ORDERED.                  ! 77
      WHERE = FOUND ! OR ORDER THE CURRENT FACE           ! 78
      L_TO_L_NEIGH (WHERE, IE) = L_TEST ! 1 of 2           ! 79
                                                    ! 80
      NEXT = COUNT ( L_TO_L_NEIGH(:, L_TEST) > 0 )        ! 81
      WHERE = NEXT+1 ! OR ORDER THE NEIGHBOR FACE         ! 82
                                                    ! 83
      IF ( L_TO_L_SUM (L_TEST) > NEXT ) &                 ! 84
        L_TO_L_NEIGH (NEXT+1, L_TEST) = IE ! 2 of 2       ! 85
      IF ( SUM_L_TO_L == FOUND ) CYCLE MAIN ! ALL         ! 86
      CYCLE RANGE ! this L_TEST element search loop      ! 87
    END IF ! NUMBER NEEDED                                ! 88
  END IF ! SHARE AT LEAST ONE COMMON NODE                 ! 89
END DO LOCAL ! over N_TEST                                ! 90
END DO RANGE ! over candidate element L_TEST             ! 91
END IF ! a possible candidate                             ! 92
END DO MAIN! over all elements                            ! 93
                                                    ! 94
IF ( NEED >= N_SPACE ) THEN ! EDGE OR FACE NEIGHBOR DATA ! 95
!      SAVE THE ELEMENT NUMBERS THAT FACE THE BOUNDARY   ! 96
  DO IE = 1, N_ELEMS                                     ! 97
    CALL GET_LT_FACES (IE, N_FACES)                      ! 98
    IF ( N_FACES > 0 ) THEN ! MIGHT BE ON THE BOUNDARY   ! 99
      IF ( ANY (L_TO_L_NEIGH (1:N_FACES, IE) == 0)) THEN !100
        ON_BOUNDARY (IE) = .TRUE.                       !101
      END IF ! ON BOUNDARY                                !102
    END IF ! POSSIBLE ELEMENT                            !103
  END DO ! OVER ELEMENTS                                  !104
END IF ! SEARCH OF FACING NEIGHBORS                      !105
END SUBROUTINE FORM_ELEMS_AT_EL                          !106

```

Figure 13.1.4b Fill the neighbor array and check boundary

# SCP_WORD	TYPICAL_VALUE	! REMARKS	[DEFAULT]
debug_scp		! Debug the SCP averaging process	[F]
face_nodes	3	! Number of shared nodes on an element face	[d]
grad_base_error		! Base error estimates on gradients only	[F]
list_el_to_el		! List elements adjacent to elements	[F]
no_scp_ave		! Do NOT get superconvergent patch averages	[F]
no_error_est		! Do NOT compute SCP element error estimates	[F]
pt_el_list		! List all the elements at each node	[F]
scp_center_only		! Use center node or element only in average	[T]
scp_center_no		! Use all elements in the patch in average	[F]
scp_deg_inc	1	! Increase patch degree by this (1 or 2)	[0]
scp_max_error	5.	! Allowed % error in energy norm	[1]
scp_neigh_el		! Element based patch, all neighbors (default)	[T]
scp_neigh_face		! Element based patch, facing neighbors	[F]
scp_neigh_pt		! Nodal based patch, all element neighbors	[F]
scp_not_once		! Scatter a node at each appearance	[F]
scp_only_once		! Scatter to a node only once per patch	[T]
scp_2nd_deriv		! Recover 2nd derivatives data also	[F]

Figure 13.1.5 Optional SCP control keywords

are sometimes called *Barlow points*. The derivation of the locations generally shows them to coincide with the Gaussian quadrature points (as illustrated here in sections 3.8 and 6.5). Here we will assume that the minimum number of quadratic points needed to properly form the element matrices have locations that correspond to the element superconvergent points, or are reasonably close to them. Thus, as we process each element to build its square matrix, we will want to save, at each quadrature point, its physical location in space and the differential operator matrix, \mathbf{B} , that will allow the accurate gradients to be computed from the local nodal solution. Looking ahead to the error estimation or other post-processing, we know that at times we will also want to have the constitutive matrix, \mathbf{E} , so we will also save it. Note that we are allowing for different, but compatible, element shapes in the mesh (and patches) and they would require different integration rules.

Now we should look ahead to how the above data are to be recovered in the SCP section of the code. The main observation is that, for an unstructured mesh, the element numbers for the elements adjacent to a particular node or element are totally random. While we have a straight-forward way to save the above data in a sequential fashion, we need to recover the element data in a random fashion. Thus, we either need to build a database that allows random access or we must decide to re-compute the data in each element of each patch. The latter is too expensive, so we must select the new database option. We could select linked lists, or a tree structure, but there is a simpler way. Fortran has always had a feature known as a "direct access" file that allows the user to randomly recover or change data. The actual data structure employed is left up to the group that writes the compiler, and is mainly hidden from the user. However, the user must declare the "record number" of the data set to be recovered or changed. Likewise, the record number of each data set must be given as the data are saved to the random access file. This means that some logical way will be needed to create a unique number for each record at any quadrature point in the mesh.


```

SUBROUTINE POST_PROCESS_GRADS (NODES, DD, ITER) ! 1
! * * * * * ! 2
!   SAVE ELEMENT GRADIENTS AS SCP INPUT RECORDS ! 3
! * * * * * ! 4
Use System_Constants ! for L_S_TOT, N_D_FRE, NOD_PER_EL, ! 5
! N_L_TYPE, N_PRT, THIS_EL, U_FLUX ! 6
Use Elem_Type_Data ! for PT (LT_PARM, LT_QP), WT (LT_QP), ! 7
! G (LT_GEOM, LT_QP), DLG (LT_PARM, LT_GEOM, LT_QP), ! 8
! H (LT_N), DLH (LT_PARM, LT_N, LT_QP), C (LT_FREE), ! 9
! S (LT_FREE, LT_FREE), ELEM_NODES (LT_N), D (LT_FREE) !10
Use Interface_Header ! for GET_ELEM_* !11
IMPLICIT NONE !12
REAL(DP), INTENT(IN) :: DD (N_D_FRE) !13
INTEGER, INTENT(IN) :: NODES (L_S_TOT, NOD_PER_EL), ITER !14
INTEGER :: IE, LT ! Loops, element type !15
!16
! D = NODAL PARAMETERS ASSOCIATED WITH AN ELEMENT !17
! DD = ARRAY OF SYSTEM DEGREES OF FREEDOM !18
! INDEX = SYSTEM DOF NOS ASSOCIATED WITH ELEMENT !19
! ITER = CURRENT ITERATION NUMBER !20
! ELEM_NODES = THE NOD_PER_EL INCIDENCES OF THE ELEMENT !21
! NOD_PER_EL = NUMBER OF NODES PER ELEMENT !22
! N_D_FRE = TOTAL NUMBER OF SYSTEM DEGREES OF FREEDOM !23
! N_L_TYPE = NUMBER OF DIFFERENT ELEMENT TYPES USED !24
! N_ELEMS = NUMBER OF ELEMENTS IN SYSTEM !25
! NODES = ELEMENT INCIDENCES OF ALL ELEMENTS !26
! U_FLUX = BINARY UNIT TO STORE GRADIENTS OR FLUXES !27
!28
LT = 1 ! INITIALIZE ELEMENT TYPES !29
WRITE (N_PRT, "(/, 'BEGIN SCP SAVE, ITER =', I4)") ITER !30
!31
!--> LOOP OVER ELEMENTS !32
DO IE = 1, N_ELEMS ! for elements, boundary segments !33
CALL SET_THIS_ELEMENT_NUMBER (IE) ! Set THIS_EL !34
!35
! VALIDATE ELEMENT TYPE !36
IF ( N_L_TYPE > 1) LT = L_TYPE (IE) ! GET TYPE !37
IF ( LT /= LAST_LT ) THEN ! this is a new type !38
CALL SET_ELEM_TYPE_INFO (LT) ! Set controls !39
END IF ! a new element type !40
!41
! RECOVER ELEMENT DEGREES OF FREEDOM !42
ELEM_NODES = GET_ELEM_NODES (IE, LT_N, NODES) !43
INDEX = GET_ELEM_INDEX (LT_N, ELEM_NODES) !44
D = GET_ELEM_DOF (DD) ! Get all nodal dof !45
!46
!--> USE DOF TO RECOVER FLUXES, LIST, SAVE FOR SCP !47
IF ( USE_EXACT_FLUX ) THEN !48
CALL LIST_ELEM_AND_EXACT_FLUXES (U_FLUX, IE) !49
ELSE !50
CALL LIST_ELEM_FLUXES (U_FLUX, IE) !51
END IF ! an exact solution is known !52
END DO ! over all elements !53
END SUBROUTINE POST_PROCESS_GRADS !54

```

Figure 13.1.6 Preparing data for averaging or post-processing

For a mesh with a single element type and a single integration rule, we could write a simple equation for the record number. Here we are allowing a mixture of element types and quadrature rules, so we store the record number at each quadrature point in an

integer array sized for the maximum number of elements and the maximum number of quadrature points per element. The record numbers are created sequentially as the element matrices are integrated. A file structure, SCP_RECORD_NUMBER, is supplied for randomly recovering the integer record number at any integration point in any element. Like any other file used in a program, a random access file must be opened. It is opened as a DIRECT access file of UNFORMATTED, or binary, records to minimize storage. We must also declare the length of the data records. It is actually hardware-dependent, so F90 includes an intrinsic function, INQUIRE (IOLENGTH), that will compute the record length given a list of variables and/or arrays to be included in each record. The unit number assigned to the random access file holding the SCP records is given the variable name U_SCPR.

As mentioned above the SCP process can be used to determine the average nodal fluxes and to use them to compute the element error estimates. The general outline of the process is as follows:

1. Preliminary
 - a. Build a list of element neighbors.
 - b. Open the sequential file unit U_FLUX to receive element data related to flux calculations. Those data can also be used for optional post-processing.
 - c. Compute the record length necessary to store the coordinates and flux components at a point.
 - d. Open the random access file unit U_SCPR that will receive the quadrature point coordinates and flux components.
2. Element Matrices Generation Loop
 - a. For each element save its number of integration points to file unit U_FLUX.
 - b. Within the numerical integration loop of the element sequentially save the arrays **XYZ**, **E**, and **B** at each point so that the gradients and/or flux components can be found at the point.
 - c. When all elements have been processed rewind the file U_FLUX to its beginning.

3. Flux Calculations and Saving Them for Averaging

After the solution has been obtained it is possible to compute the flux (and gradient) components within each element so that they can be smoothed to nodal values. The element flux calculation is done in subroutine POST_PROCESS_GRADS, as detailed in Fig. 13.1.6. First, the SCP record number is set to zero. Next, each element is processed in a loop:

- a. Recover the element type;
- b. Gather the nodal degrees of freedom of the element;
- c. Read the number of quadrature points in the element from U_FLUX;
- d. Quadrature Point Loop

For each integration point, in routine LIST_ELEM_FLUXES, sequentially recover the **XYZ**, **E**, and **B** arrays from U_FLUX. Multiply **B** by the element *dof* to get the gradients or strains at the point, and then multiply those by the


```

!   IS THIS AN ELEMENT, BOUNDARY, OR ROBIN SEGMENT ?           !49
IF ( IS_ELEMENT ) THEN ! ELEMENT RESULTS                       !50
                                                                    !51
    READ (N_TAPE, IOSTAT = EOF) N_IP ! # INTEGRATION POINTS    !52
                                                                    !53
!-->    READ COORDS, CONSTITUTIVE, AND DERIVATIVE MATRIX        !54
    DO J = 1, N_IP ! OVER ALL INTEGRATION POINTS              !55
        READ (N_TAPE, IOSTAT = EOF) XYZ, E, B                 !56
                                                                    !57
!        CALCULATE DERIVATIVES, STRAIN = B * D                 !58
        STRAIN (1:N_R_B) = MATMUL (B, D)                       !59
                                                                    !60
!        FLUX FROM CONSTITUTIVE DATA                           !61
        STRESS (1:N_R_B) = MATMUL (E, STRAIN (1:N_R_B))       !62
                                                                    !63
!-->    PRINT COORDINATES AND FLUX AT THE POINT                 !64
        WRITE (N_PRT, '(I7, I3, 10(ES12.4))') &              !65
            IE, J, XYZ, STRESS (1:N_R_B)                       !66
                                                                    !67
!-->    STORE FLUX RESULTS TO BE PLOTTED LATER, IF USED        !68
        IF (U_PLT4 > 0) WRITE (U_PLT4, '( (10(1PE13.5)) )') & !69
            XYZ, STRESS (1:N_R_B)                              !70
        IF (N_TAPE5 > 0) WRITE (N_TAPE5, '( (10(1PE13.5)) )') & !71
            XYZ, STRAIN (1:N_R_B)                              !72
                                                                    !73
!        SAVE COORDINATES & FLUX FOR SCP FLUX AVERAGING       !74
        IF ( U_SCPR > 0 ) THEN ! SCP recovery is active        !75
            RECORD_NUMBER = RECORD_NUMBER + 1                 !76
            SCP_RECORD_NUMBER (IE, J) = RECORD_NUMBER         !77
                                                                    !78
            IF ( GRAD_BASE_ERROR ) THEN ! User override       !79
                WRITE (U_SCPR, REC = RECORD_NUMBER) &        !80
                    XYZ, STRAIN (1:SCP_FIT)                   !81
            ELSE ! Usual case                                   !82
                WRITE (U_SCPR, REC = RECORD_NUMBER) &        !83
                    XYZ, STRESS (1:SCP_FIT)                   !84
            END IF ! GRAD VS FLUX IS DESIRED                   !85
        END IF ! SCP RECOVERY                                  !86
                                                                    !87
    END DO ! OVER INTEGRATION POINTS                           !88
END IF ! ELEMENT OR BOUNDARY SEGMENT OR MIXED BOUNDARY        !89
CALL UPDATE_SCP_STATUS ! FLAG IF SCP DATA WERE SAVED         !90
END SUBROUTINE LIST_ELEM_FLUXES                               !91

```

Figure 13.1.7b Saving element gradient or flux data to files

constitutive array, \mathbf{E} , to get the fluxes, or stresses at the point. The element and quadrature point numbers are then printed along with their coordinates and flux, or stress, components. Lastly, the SCP database is updated by incrementing the record number by one, and then writing the coordinates and flux component arrays to the random access file, U_SCPR , as that record is to be later randomly recovered in the patch smoothing process. The above details are shown in Fig. 13.1.7.

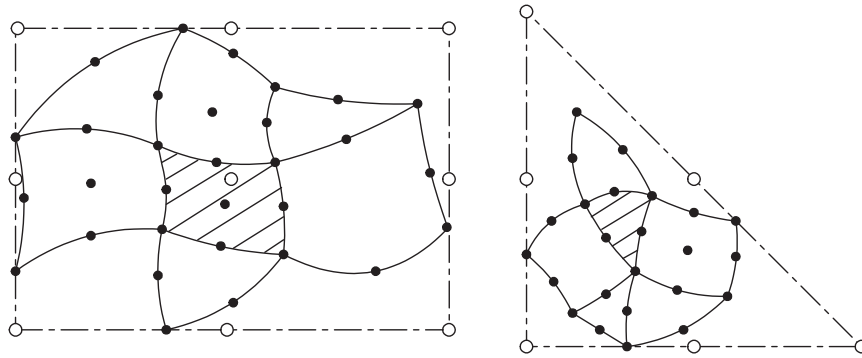


Figure 13.2.1 Bounding a group of elements with a constant Jacobian patch

13.2 SCP Nodal Flux Averaging

Having developed the above database on unit `U_SCPR` we can now average the flux components at every node in each patch and then average them for each node in the mesh. Here we assume an element based patch system for calculating the averages. In subroutine `CALC_SCP_AVE_NODE_FLUXES` we loop over every element and carry out the least squares fit in its associated patch. Looking ahead to that process recall that we must select a polynomial, \mathbf{P} , to be used in the patch. We must make a choice for that function. We might select a complete polynomial of a given degree, or a Serendipity polynomial of a given edge degree, etc. In the current implementation the default is to select that polynomial to be exactly the same as the polynomial used to interpolate the element for which the patch is being constructed. This means that we will select a constant Jacobian patch "element" that has its local axes parallel to the global axes and completely surrounds the standard elements that make up the patch. This is easily done by searching for the maximum and minimum components of all of the element nodes in the patch. Such a process is illustrated in Fig. 13.2.1 where the active element used to select the patch element type is shown crosshatched. It would also be easy to allow the user to select a patch type and degree through a keyword control input. The full details of the process are given in the source code of Fig.13.2.2 and the main points are outline below.

The least squares flux averaging process is:

- a. Zero the nodal flux array and the counter for each node.
- b. Loop over each element in the mesh:
 1. Extract its element neighbors to define the patch
 2. Find the spatial "box" that bounds the patch
 3. Find the number of quadrature points in the patch (i.e. sum the count in each element of the patch).
 4. Determine the element type and thus the patch "element" shape (line, triangle, hexahedron, etc.) and the corresponding patch polynomial degree.
 5. Allocate storage for the least squares fit arrays.


```

! SCP_N      = NUMBER OF NODES PER PATCH           ! 56
! SCP_RECORD_NUMBER = SCP DIRECT ACCESS RECORD LOCATOR ! 57
! U_SCP      = SUPER_CONVERGENT PATCH RECOVERY UNIT ! 58
! X          = COORDINATES OF SYSTEM NODES         ! 59
! XYZ       = SPACE COORDINATES AT A POINT         ! 60
! XYZ_MAX   = UPPER BOUNDS FOR SCP GEOMETRY        ! 61
! XYZ_MIN   = LOWER BOUNDS FOR SCP GEOMETRY        ! 62
!           ! 63
LT=1 ; LAST_LT=0 ; SCP_COUNTS=0 ; SCP_AVERAGES=0 ! 64
!           ! 65
IF ( N_PATCH == 0 ) THEN ! No data supplied ! 66
  PRINT *, 'NO PATCHS GIVEN, SKIPPING AVERAGES'; RETURN ! 67
END IF ! 68
!           ! 69
DO IP = 1, N_PATCH ! LOOP OVER EACH PATCH ! 70
  MEMBERS = 0 ; POINTS = 0 ! INITIALIZE ! 71
!           ! 72
!   ELEMENT OR NODAL CENTERED PATCH TYPE ! 73
  IF ( .NOT. SCP_NEIGH_PT ) THEN ! ELEMENT BASED ! 74
!   GET ELEMENT NEIGHBORS TO DEFINE THE PATCH ! 75
    MEMBERS = ( / IP, L_NEIGH (:, IP) / ) ! 76
  ELSE ! NODAL BASED PATCH OF ELEMENTS ! 77
!   GET ELEMENT NEIGHBORS TO DEFINE THE PATCH ! 78
    MEMBERS = L_NEIGH (:, IP) ! 79
  END IF ! PATCH BASIS ! 80
!           ! 81
  L_IN_PATCH = COUNT ( MEMBERS > 0 ) ! 82
  IF ( L_IN_PATCH <= 1 ) CYCLE ! TO AN ACTIVE PATCH ! 83
!           ! 84
!   FIND TYPE OF SCP NEEDED HERE, VERIFY GEOMETRY ! 85
  CALL DETERMINE_SCP_BOUNDS (L_IN_PATCH, MEMBERS, & ! 86
    NODES, X, XYZ_MIN, XYZ_MAX, POINTS) ! 87
!           ! 88
  IF ( PATCH_ALLOC_STATUS ) THEN ! DEALLOCATE ARRAYS ! 89
    DEALLOCATE ( PATCH_WRK ) ; DEALLOCATE ( PATCH_SQ ) ! 90
    DEALLOCATE ( PATCH_FIT ) ; DEALLOCATE ( PATCH_DAT ) ! 91
    DEALLOCATE ( PATCH_P ) ; PATCH_ALLOC_STATUS=.FALSE. ! 92
  END IF ! STATUS CHECK ! 93
!           ! 94
!   ALLOCATE NEXT SET OF LOCAL PATCH RELATED ARRAYS ! 95
  ALLOCATE ( PATCH_P (POINTS, SCP_N ) ) ! 96
  ALLOCATE ( PATCH_DAT (POINTS, SCP_FIT) ) ! 97
  ALLOCATE ( PATCH_FIT (SCP_N , N_R_B ) ) ! 98
  ALLOCATE ( PATCH_SQ (SCP_N , SCP_N ) ) ! 99
  ALLOCATE ( PATCH_WRK (SCP_N ) ) !100
  PATCH_ALLOC_STATUS = .TRUE. !101
!           !102
!   ZERO PATCH WORKSPACE AND RESULTS ARRAYS !103
  PATCH_P = 0.d0 ; PATCH_DAT = 0.d0 ; PATCH_FIT = 0.d0 !104
  PATCH_SQ = 0.d0 ; PATCH_WRK = 0.d0 !105
!           !106

```

Figure 13.2.2b Establish bounds and dynamic memory for each patch

```

!   PREPARE LEAST SQUARES FIT MATRICES                               !107
LAST_LT = 0 ; ROW = 0                                               !108
DO IL = 1, L_IN_PATCH                                               !109
  LM = MEMBERS (IL)                                                 !110
                                                                    !111
  ! GET ELEMENT TYPE NUMBER                                         !112
  IF (N_L_TYPE > 1) LT=L_TYPE (LM)                                  !113
  IF ( LT /= LAST_LT ) THEN                                         !114
    CALL GET_ELEM_TYPE_DATA (LT)  ! TYPE CONTROLS                 !115
    LAST_LT = LT                                                    !116
  END IF ! a new element type and scp type                          !117
                                                                    !118
  DO IQ = 1, LT_QP                                                  !119
    ROW = ROW + 1                                                   !120
                                                                    !121
  !   RECOVER EACH FLUX VECTOR TO FOR LEAST SQ FIX                 !122
  REC_LM_IQ = SCP_RECORD_NUMBER (LM, IQ) ! REC NUMBER              !123
                                                                    !124
  !   GET GAUSS PT COORD & FLUX SAVED IN LIST_ELEM_FLUXES         !125
  READ (U_SCP,REC=REC_LM_IQ,IOSTAT=SCP_STAT) XYZ, FLUX             !126
  SELECT CASE ( SCP_STAT ) ! for read status                       !127
    CASE (:-1)                                                       !128
      STOP 'EOR OR EOF, CALC_SCP_AVE_NODE_FLUXES'                 !129
    CASE (1:)                                                         !130
      PRINT *, 'MEMBER ELEMENT = ', LM, ' AT POINT ', IQ          !131
      PRINT *, 'REC_LM_IQ      = ', REC_LM_IQ                     !132
      STOP 'BAD SCP_STAT, CALC_SCP_AVE_NODE_FLUXES'               !133
    CASE DEFAULT ! NO READ ERROR                                     !134
  END SELECT ! RANDOM ACCESS READ ERROR                             !135
                                                                    !136
  !   CONVERT IQ XYZ TO LOCAL PATCH POINT                           !137
  POINT = GET_SCP_PT_AT_XYZ (XYZ, XYZ_MIN, XYZ_MAX)                 !138
                                                                    !139
  !   EVALUATE PATCH INTERPOLATION AT LOCAL POINT                  !140
  IF ( .NOT. SCP_SCAL_ALLOC ) CALL &                               !141
    ALLOCATE_SCP_INTERPOLATIONS                                     !142
  CALL GEN_ELEM_SHAPE (POINT, SCP_H, SCP_N, N_SPACE, ONE)           !143
                                                                    !144
  !   INSERT FLUX & INTERPOLATIONS INTO PATCH MATRICES            !145
  PATCH_DAT (ROW, 1:N_R_B) = FLUX (:)                               !146
  PATCH_P   (ROW, :)      = SCP_H (:)                               !147
  END DO ! FOR EACH IQ FLUX VECTOR                                  !148
                                                                    !149
  END DO ! FOR EACH IL PATCH MEMBER                                 !150
! ASSEMBLY OF PATCH COMPLETED                                     !151
                                                                    !152
!   VALIDATE CURRENT PATCH                                         !153
IF ( POINTS < SCP_N ) THEN                                         !154
  PRINT *, 'WARNING: SKIPPING PATCH ', &                          !155
    IP, ' WITH ONLY ', POINTS, ' EQUATIONS'                       !156
  CYCLE ! TO NEXT PATCH                                           !157
END IF ! INSUFFICIENT DATA                                       !158

```

Figure 13.2.2c Build the least squares fit in each patch


```

!      USE SINGULAR VALUE DECOMPOSITION SOLUTION METHOD           !159
CALL SVDC_FACTOR (PATCH_P,POINTS,SCP_N,PATCH_WRK,PATCH_SQ) !160
WHERE ( PATCH_WRK < EPSILON(1.d0) ) PATCH_WRK = 0.D0           !161
                                                                !162
DO FIT = 1, N_R_B      ! LOOP FOR EACH FLUX COMPONENT           !163
  CALL SVDC_BACK_SUBST (PATCH_P, PATCH_WRK, PATCH_SQ, &         !164
    POINTS, SCP_N, PATCH_DAT (:, FIT), &                       !165
    PATCH_FIT (:, FIT))                                         !166
END DO ! FOR FLUXES      COMPONENTS                             !167
                                                                !168
!  INTERPOLATE AVERAGES TO ALL NODES IN THE PATCH. SCATTER     !169
!  PATCH NODAL AVERAGES TO SYSTEM NODES, INCREMENT COUNTS     !170
                                                                !171
IF (DEBUG_SCP .AND. IP==1) PRINT *, 'AVERAGING FLUXES'        !172
CALL EVAL_SCP_FIT_AT_PATCH_NODES (IP, NODES, X,                & !173
  L_IN_PATCH, MEMBERS, XYZ_MIN, XYZ_MAX, PATCH_FIT, &         !174
  SCP_AVERAGES, SCP_COUNTS)                                    !175
                                                                !176
! NOTE: use Loubignac iteration here for new solution          !177
                                                                !178
!      DEALLOCATE LOCAL PATCH RELATED ARRAYS                    !179
IF (SCP_SCAL_ALLOC) CALL DEALLOCATE_SCP_INTERPOLATIONS         !180
DEALLOCATE (PATCH_WRK) ; DEALLOCATE (PATCH_SQ )             !181
DEALLOCATE (PATCH_FIT) ; DEALLOCATE (PATCH_DAT)             !182
DEALLOCATE (PATCH_P ) ; PATCH_ALLOC_STATUS=.FALSE.          !183
                                                                !184
END DO ! FOR EACH IP PATCH IN MESH                             !185
                                                                !186
!  FINALLY, AVERAGE FLUXES FOR EACH NODAL HIT COUNT           !187
DO FIT = 1, MAX_NP ! FOR ALL NODES                             !188
  IF ( SCP_COUNTS (FIT) /= 0 ) THEN ! ACTIVE NODE              !189
    SCP_AVERAGES (FIT, :) = SCP_AVERAGES (FIT, :) &            !190
      / SCP_COUNTS (FIT)                                       !191
  ELSE ! could skip since initialized                           !192
    SCP_AVERAGES (FIT, :) = 0.D0                                !193
  END IF                                                         !194
END DO ! FOR (AN UNWEIGHTED) AVERAGE                          !195
                                                                !196
!  REPORT AVERAGED MAX & MIN VALUES AND LOCATIONS            !197
CALL MAX_AND_MIN_SCP_AVE_F90 (SCP_AVERAGES)                   !198
END SUBROUTINE CALC_SCP_AVE_NODE_FLUXES                       !199

```

Figure 13.2.2d Factor each patch then average at all nodes

6. Set the fit matrix row number to zero.
7. For each element in the patch loop over the following steps:
 - A. Find its type and quadrature rule
 - B. Loop over each of its quadrature points
 - 1) Increment the row number by one.
 - 2) Use the element number and quadrature point number pair as subscripts in the SCP_RECORD_NUMBER function to recover the random record number for that point.
 - 3) Read the physical coordinates and flux components from random access file U_SCPR by using that record number.
 - 4) Use the constant Jacobian of the patch to convert the physical location to the corresponding non-dimensional coordinates in the

- patch. Note that this helps reduce the numerical ill-conditioning that is common in a least squares fit process.
- 5) Evaluate the patch interpolation polynomial at the local point (by utilizing the standard element interpolation library). Insert it into the left hand side of this row of the coefficient matrix.
 - 6) Substitute the flux components into the right hand side data matrix, in the same row. Of course the number of columns on the right hand side is the same as the number of flux components. This is also the size of the patch result matrix, \mathbf{a} , to be computed.
8. Having completed the loop over all the elements in this patch we now have the rectangular arrays cited in Eq. 12.28 but we have not computed their actual matrix products as shown in Eq. (12.29). While that equation is the standard way to describe a least squares fit we do not actually use that process. Instead, we try to avoid possible numerical ill-conditioning by using an equivalent but more powerful process called the singular value decomposition algorithm [8]. That process first factors the associated patch square matrix (in subroutine SVDC_FACTOR) and then recovers the rectangular array of local continuous patch nodal flux values (with subroutine SVDC_BACK_SUBST). However, we want smooth flux values at the actual nodes of the elements, not values at the patch nodes. Thus, for the elements in question we need to interpolate the patch results back to the system nodes contained within the current patch.
- 9) Loop over nodes in this patch:
- a) Use the constant patch Jacobian to convert the node coordinates to non-dimensional coordinates of the patch
 - b) Evaluate the patch interpolation matrix, \mathbf{SCP}_H , at each node point. It is usually the same as the core element interpolation functions \mathbf{H} .
 - c) Compute the flux components at the node by the matrix product of \mathbf{SCP}_H and the continuous gradients at the patch nodes, \mathbf{a} .
 - d) Increment the nodal counter for patch contributions by one and scatter the node flux components to the rectangular system flux nodal array.
- c. Optional Improvement of the Solution

At this stage in the SCP process one can use the least square smoothed gradients in this patch to get a locally improved solution value estimate at all of the patch's interior nodes. However, one may want to just do so for the single parent element about which the patch was constructed. The algorithm is a form of the Loubignac [7] iterative process:

- 1) Read or reform element matrices, \mathbf{S}_e and \mathbf{C}_e , here;
- 2) Use a higher order quadrature rule to form the equilibrating vector

$$\mathbf{V}_e = \int_{\Omega^e} \mathbf{B}^* (\boldsymbol{\sigma}^* - \hat{\boldsymbol{\sigma}}) d\Omega ;$$

- 3) Assemble the elements in the patch into a local linear system:

$$\mathbf{S}^* \boldsymbol{\phi}_{\text{new}} = \mathbf{C} - \mathbf{V} ;$$

- 4) Apply the previous solution as essential boundary conditions at all nodes on the patch boundary;
 - 5) Solve for the new interior node values (always a small system but especially small for a patch of elements around a single node as in original ZZ patch paper). Call them ϕ_e^* .
 - 6) Compute norm of $\phi_e - \phi_e^*$ to use as an additional term in the final error estimator.
- d. Final nodal flux average.

As shown in Fig. 13.2.3, most nodes are associated with more than one patch. Having processed every patch for the mesh each node has now received as many nodal flux estimates as there were patches that contained that node. We finalize the nodal flux values by simply dividing each node's flux components sums by that integer counter, print the result, and re-save them in the rectangular array SCP_AVERAGES for use by the element error estimator or other user-defined post-processing.

13.3 Computing the SCP Element Error Estimates

For a homogeneous domain, or sub-domain, the above nodal averaging process provides a continuous flux approximation that should be much closer to the true solution than the element discontinuous fluxes. Thus, it is reasonable to base the element error estimator on the average nodal fluxes from the SCP process. Basically, we will want to integrate the difference between the spatial distributions of the two flux estimates so that we can calculate the error norms of interest in each element. Then we will sum those scalar values over all elements so that we can establish the relative errors and how they compare to the allowed value specified by the user. Of course, we will evaluate the

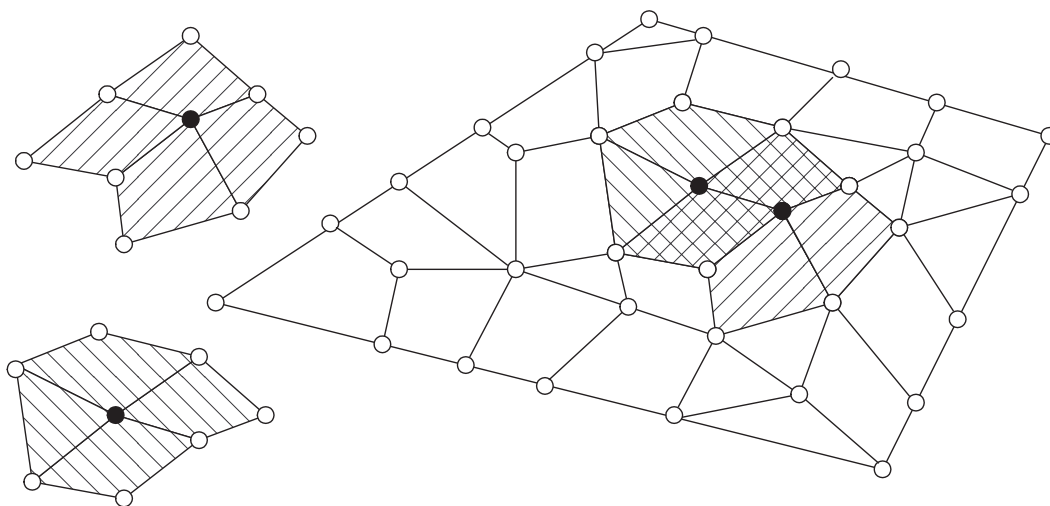


Figure 13.2.3 Overlapping patches give multiple node estimates

element norms by numerical integration. This will require a higher order quadrature rule than the one needed to evaluate the element square matrix because the interpolation function, \mathbf{P} (which is usually \mathbf{H}), is of higher polynomial degree than the \mathbf{B} matrix (which contains the derivatives of \mathbf{H}) used in forming the element square matrix.

Subroutine SCP_ERROR_ESTIMATES implements the major steps outlined below.

1. Preliminary Setup
 - a. Initialize all of the norms to zero.
 - b. If the mesh has a constant constitutive matrix, \mathbf{E} , then recover it and invert it once for later use in calculating the energy norm.
2. Loop over all elements in the mesh:
 - a. Recover the element type (shape, number of nodes, quadrature rule for \mathbf{B} , etc.)
 - b. Determine the quadrature rule to integrate the \mathbf{P} array (here the \mathbf{H} array), allocate storage for that array to be pre-computed at each quadrature point, and then fill those arrays.
 - c. Extract the element's node numbers, coordinates, and *dof*.
 - d. At each node on the element gather the continuous nodal flux components from the system SCP averages, $\mathbf{a}_e \subset \mathbf{a}$.
 - e. Numerical integration loop over the element to form element norms and increment system norms:
 1. Recover the \mathbf{H} array at the point and its local derivatives.
 2. Obtain the physical coordinates, Jacobian and its inverse.
 3. If the L_2 norm of the solution is desired, interpolate for the value at the point. Increment the L_2 norm integral. If the exact solution has been provided, then compute its norm also.
 4. Compute the physical gradients of the original finite element solution. Form the \mathbf{B} matrix at the point for the current application.

$$\boldsymbol{\varepsilon} = \mathbf{B} \boldsymbol{\phi}$$

If the constitutive array, \mathbf{E} , is smoothly varying, then we could evaluate it at this point (and compute its inverse). Otherwise, we employ the \mathbf{E} matrix saved in the preliminary setup. Now we recover the standard element flux estimate by matrix multiplication

$$\hat{\boldsymbol{\sigma}} = \mathbf{E} \boldsymbol{\varepsilon} .$$

We can increment the L_2 norm of this term if desired.

5. Now we are ready to recover the continuous flux values and approximate the stress error. We simply carry out the matrix product of the interpolation functions, \mathbf{H} , and the nodal fluxes, \mathbf{a} , at the quadrature point.

$$\boldsymbol{\sigma}^* = \mathbf{H} \mathbf{a}$$

The difference between these components and those from the previous step are formed to define the stress error

$$\mathbf{e}_\sigma = \boldsymbol{\sigma}^* - \hat{\boldsymbol{\sigma}} .$$

If desired the square of this term (its dot product with itself) is obtained for its

increment to the L_2 stress norm.

6. In this implementation we almost always use the flux error to compute the error energy norm, so at this stage we form the related triple matrix product, $\mathbf{e}_\sigma^T \mathbf{E}^{-1} \mathbf{e}_\sigma$, and increment the quadrature point contribution to the element norm

$$\|\mathbf{e}^e\|^2 = \sum_q^{NQ} (\boldsymbol{\sigma}^* - \hat{\boldsymbol{\sigma}})_q \mathbf{E}_q^{-1} (\boldsymbol{\sigma}^* - \hat{\boldsymbol{\sigma}})_q .$$

If the user has supplied an expression for the exact flux components, then they are evaluated at the physical coordinates, and the corresponding L_2 and error energy norms are updated for later comparisons and to find the effectivity index.

Having incremented all of the element norms, they are complete at the end of this quadrature loop for the current element. The active element norm values are then added to the current values of the corresponding system norms. At times we also want to use the element and system volume measures so that we can get some norm volumetric averages. Thus, the determinant of the Jacobian at the above points are also used to obtain the element volumes so that they are available for these optional calculations.

Upon completing the loop over all elements we have the element norms, the element volume, the system norms, and the system volume. The allowed error energy is obtained from the product of the strain energy norm and the user input value of the allowed percentage error, η (keyword input value *scp_allow_error_%*). That number is used, in turn, to evaluate two allowed error densities in dividing by the square root of the number of elements and the square root of the volume to yield mean element and volumetric references, respectively. One of these reference values will be used to rank the relative error measures in each element. The system norm values are printed along with the two allowable reference values for the energy error and the system volume. For each element we will list the element error norm, its percentage of the strain energy norm, and a refinement parameter for that element. The element error energy norms are summed to get the total energy in the error to compare to the total strain energy norm and the allowed percentage of error. Here, the refinement parameter is based on the volumetric error density, so for element j the refinement parameter is

$$\xi_j = (\|\mathbf{e}_j\|/\sqrt{\Omega_j})/(\eta\|\mathbf{e}\|/\sqrt{\Omega})$$

or

$$\xi_j = \frac{\|\mathbf{e}_j\|}{\eta\|\mathbf{e}\|} \left(\frac{\Omega}{\Omega_j} \right)^{\frac{1}{2}} . \quad (13.1)$$

Here it is informative to note that for a uniform mesh all the NE element volumes are constant with a value of

$$\Omega_j = \Omega/NE$$

and the refinement indicator becomes the same as originally employed by Zienkiewicz and Zhu:

$$\xi_j = \frac{\|\mathbf{e}_j\|}{\eta\|\mathbf{e}\| \sqrt{NE}} . \quad (13.2)$$

By combining such an indicator with interpolation error analysis, one can predict the

desired element size or polynomial degree. For each element, i , we define the ratio ξ to indicate needed refinement when $\xi_i > 1$ and de-refinement when $\xi_i < 1$. The coding details associated with these steps are in the single subroutine but are broken out into major segments in Fig. 13.3.1.

13.4 H-Adaptivity

If h_i is the current element size, then asymptotic convergence rate estimates suggest that the new element size should be smaller than $h = h_i / \xi^{1/p}$ where p is the polynomial order of the interpolation for ϕ .

13.5 HP-Adaptivity

While asymptotic analysis makes it easy to estimate the new element size, h , when the polynomial degree, p , is held constant things become less clear when p , or p and h , are allowed to change. Various heuristic rules are developed for such a process. For example, p must be an integer and we usually want it to increase slowly, since it is expensive, so we might restrict the new value to be only one degree higher or lower.

13.6 Second Derivative Recovery *

There are times when one is also interested in the estimates of the second derivatives of the solution. Examples include the application of the Streamline Upwind Petrov Galerkin (SUPG) method for advection-diffusion problems and the inclusion of "stabilization" (or governing PDE residual) terms in the solution of the Navier-Stokes equations. If one is employing high-order interpolation elements, one could proceed with direct estimates of the second derivatives at the element level. Of course, we would expect a decrease in accuracy compared to the element gradient estimates. Assuming the use of parametric elements, we need to employ the Jacobian. Recall that the Jacobian defines the mapping from the parametric space the physical space. In two dimensions:

$$\begin{Bmatrix} \frac{\partial}{\partial r} \\ \frac{\partial}{\partial s} \end{Bmatrix} = \begin{bmatrix} \frac{\partial x}{\partial r} & \frac{\partial y}{\partial r} \\ \frac{\partial x}{\partial s} & \frac{\partial y}{\partial s} \end{bmatrix} \begin{Bmatrix} \frac{\partial}{\partial x} \\ \frac{\partial}{\partial y} \end{Bmatrix}.$$

Continuing this process to relate second parametric derivatives to second physical derivatives involves products and derivatives of the Jacobian array. For a two-dimensional mapping:

```

SUBROUTINE SCP_ERROR_ESTIMATES (NODES, X, SCP_AVERAGES, DOF_SYS,& ! 1
                                ELEM_ERROR_ENERGY, & ! 2
                                ELEM_REFINEMENT, ERR_MAX) ! 3
! * * * * * ! 4
! USE INTEGRAL OF DIFFERENCE BETWEEN THE RECOVERED AVERAGE NODAL ! 5
! FLUXES AND ORIGINAL ELEMENT FLUXES TO ESTIMATE ELEMENT ERROR ! 6
! IN THE ENERGY NORM ! 7
! * * * * * ! 8
! Note: Debug options, 2nd derivative options, saving to ! 9
!       plotter files are not shown here to save space. ! 10
!       See source library for full details. ! 11
Use System_Constants ! for MAX_NP, NEIGH_L, N_ELEMS, N_QP, ! 12
                        ! SCP_FIT, U_SCP, SKIP_ERROR ! 13
Use Select_Source ! 14
Use Elem_Type_Data ! for: ! 15
  ! LT_FREE, LT_GEOM, LT_N, LT_PARM, LT_QP, ELEM_NODES (LT_N),& ! 16
  ! COORD (LT_N, N_SPACE), GEOMETRY (LT_GEOM, N_SPACE), & ! 17
  ! C (LT_FREE), D (LT_FREE), S (LT_FREE, LT_FREE), & ! 18
  ! DLG (LT_PARM, LT_GEOM), DLG_QP (LT_PARM, LT_GEOM, LT_QP) & ! 19
  ! DLH (LT_PARM, LT_N), DLH_QP (LT_PARM, LT_N, LT_QP), & ! 20
  ! DLV (LT_PARM, LT_FREE), DLV_QP (LT_PARM, LT_FREE, LT_QP), & ! 21
  ! G (LT_GEOM), G_QP (LT_GEOM, LT_QP), H_QP (LT_N, LT_QP), & ! 22
  ! V (LT_FREE), V_QP (LT_FREE, LT_QP), H (LT_N), & ! 23
  ! PT (LT_PARM, LT_QP), WT (LT_QP), D2LH (N_2_DER, LT_N) ! 24
Use SCP_Type_Data ! 25
Use Interface_Header ! 26
Use Geometric_Properties ! 27
  IMPLICIT NONE ! 28
  INTEGER, INTENT (IN) :: NODES (L_S_TOT, NOD_PER_EL) ! 29
  REAL(DP), INTENT (IN) :: X (MAX_NP, N_SPACE) ! 30
  REAL(DP), INTENT (IN) :: SCP_AVERAGES (MAX_NP, SCP_FIT) ! 31
  REAL(DP), INTENT (IN) :: DOF_SYS (N_D_FRE) ! 32
  REAL(DP), INTENT (OUT) :: ELEM_ERROR_ENERGY (N_ELEMS) ! 33
  REAL(DP), INTENT (OUT) :: ELEM_REFINEMENT (N_ELEMS) ! 34
  REAL(DP), INTENT (OUT) :: ERR_MAX ! 35
  REAL (DP), PARAMETER :: ZERO = 0.d0 ! 36
! 37
  INTEGER :: IE, IN, IQ, LT, QP_LT, LOC_MAX (1) ! 38
  INTEGER :: I_ERROR ! /= 0 IFF INVERSION OF E FAILS ! 39
  REAL (DP) :: GLOBAL_ERROR_ENERGY, NEAR_ZERO ! 40
  REAL (DP) :: GLOBAL_FLUX_NORM, GLOBAL_FLUX_ERROR ! 41
  REAL (DP) :: GLOBAL_FLUX_RMS, GLOBAL_SOLUTION_L2 ! 42
  REAL (DP) :: GLOBAL_SOLUTION_ERR, VOL ! 43
  REAL (DP) :: GLOBAL_STRAIN_ENERGY, STRAIN_ENERGY_NORM ! 44
  REAL (DP) :: ELEM_STRAIN_ENERGY, ALLOWED_ERROR ! 45
  REAL (DP) :: ALLOWED_ERR_DENSITY, ALLOWED_ERR_PER_EL ! 46
  REAL (DP) :: ELEM_FLUX_NORM, ELEM_FLUX_ERROR ! 47
  REAL (DP) :: ELEM_FLUX_RMS, ELEM_SOLUTION_L2 ! 48
  REAL (DP) :: ELEM_SOLUTION_ERR, EL_ERR_ENERGY ! 49
  REAL (DP) :: GLOBAL_H1_NORM, GLOBAL_H2_NORM ! 50
  REAL (DP) :: GLOBAL_H1_ERROR, GLOBAL_H2_ERROR ! 51
  REAL (DP) :: ELEM_H1_NORM, ELEM_H2_NORM ! 52
  REAL (DP) :: ELEM_H1_ERROR, ELEM_H2_ERROR ! 53
  REAL (DP) :: EXACT_H1_NORM, EXACT_H2_NORM ! 54
  REAL (DP) :: EXACT_H1_ERROR, EXACT_H2_ERROR ! 55

```

Figure 13.3.1a Interface and data for error estimates

```

REAL (DP) :: EXACT_SOL_L2,          EX_ERR_ENERGY          ! 56
REAL (DP) :: EXACT_FLUX_ERROR,      EXACT_FLUX_NORM      ! 57
REAL (DP) :: EXACT_STRAIN_ENERGY,   EXACT_ERR_ENERGY     ! 58
REAL (DP) :: DET, DET_WT, TEMP, TEST, SCP_VOLUME          ! 59
REAL (DP), ALLOCATABLE :: DOF_EL (:, :) ! D RESHAPE        ! 60
!
! Automatic Arrays                                          ! 62
REAL (DP) :: AJ          (N_SPACE, N_SPACE)      ! JACOBIAN          ! 63
REAL (DP) :: AJ_INV      (N_SPACE, N_SPACE)      ! JACOBIAN INVERSE ! 64
REAL (DP) :: B           (N_R_B, N_EL_FRE)       ! DIFFERENTIAL OP  ! 65
REAL (DP) :: E           (N_R_B, N_R_B)          ! CONSTITUTIVE     ! 66
REAL (DP) :: E_INVERSE   (N_R_B, N_R_B)          ! INVERSE          ! 67
REAL (DP) :: DGH         (N_SPACE, NOD_PER_EL)   ! GRADIENT OF H    ! 68
REAL (DP) :: FLUX_LT     (N_R_B, NOD_PER_EL)     ! NODAL FLUXES    ! 69
REAL (DP) :: XYZ         (N_SPACE)               ! POINT IN SPACE   ! 70
REAL (DP) :: SIGMA_SCP   (N_R_B)                 ! SCP FLUX         ! 71
REAL (DP) :: SIGMA_HAT   (N_R_B)                 ! FEA FLUX        ! 72
REAL (DP) :: DIFF        (N_R_B)                 ! FLUX DIFFERENCE ! 73
REAL (DP) :: MEASURE     (N_ELEMS)               ! ELEMENT MEASURE  ! 74
REAL (DP) :: ELEM_ERROR_DENSITY (N_ELEMS)        ! Per UNIT VOLUME ! 75
REAL (DP) :: EXAC_ERROR_ENERGY (N_ELEMS)        ! EXACT ERR ENERGY ! 76
REAL (DP) :: SOLUTION (N_G_DOF), EXACT_SOL (N_G_DOF) ! AT PT           ! 77
REAL (DP) :: SOLUTION_ERR (N_G_DOF)              ! PT ERROR RESULT  ! 78
!
! B = GRADIENT VERSUS DOF MATRIX                        ! 80
! DGH = GLOBAL DERIVS OF SCALAR FUNCTIONS H            ! 81
! DOF_SYS = DEGREES OF FREEDOM OF THE SYSTEM           ! 82
! E = CONSTITUTIVE MATRIX, INVERSE IS E_INVERSE       ! 83
! ELEM_ERROR_ENERGY = ESTIMATED ELEMENT ERROR, % OF ENERGY NORM ! 84
! ELEM_ERROR_DENSITY = ESTIMATED ELEMENT ERROR / SQRT(MEASURE) ! 85
! ELEM_NODES = THE NOD_PER_EL INCIDENCES OF THE ELEMENT ! 86
! ELEM_REFINEMENT = INDICATOR, >1 REFINE, <1 DE-REFINE ! 87
! EXAC_ERROR_ENERGY = ENERGY IN ERROR FROM EXACT SOLUTION ! 88
! INDEX = SYSTEM DOF NUMBERS ASSOCIATED WITH ELEMENT   ! 89
! L_HOMO = 1, IF ELEMENT PROPERTIES ARE HOMOGENEOUS    ! 90
! L_S_TOT = TOTAL NUMBER OF ELEMENTS & THEIR SEGMENTS ! 91
! L_TYPE = ELEMENT TYPE NUMBER LIST                    ! 92
! LT = ELEMENT TYPE NUMBER (IF USED)                   ! 93
! LT_QP = NUMBER OF QUADRATURE PTS FOR ELEMENT TYPE    ! 94
! MAX_NP = NUMBER OF SYSTEM NODES                      ! 95
! MEASURE = ELEMENT MEASURE (GENERALIZED VOLUME)      ! 96
! NOD_PER_EL = MAXIMUM NUMBER OF NODES PER ELEMENT    ! 97
! NODES = NODAL INCIDENCES OF ALL ELEMENTS            ! 98
! N_QP = MAXIMUM NUMBER OF QUADRATURE POINTS, >= LT_QP ! 99
! N_R_B = NUMBER OF ROWS IN B AND E MATRICES           !100
! N_SPACE = DIMENSION OF SPACE                         !101
! SCP_AVERAGES = AVERAGED FLUXES AT ALL NODES IN MESH !102
! SCP_FIT = NUMBER IF TERMS BEING FIT, OR AVERAGED    !103
! SCP_VOLUME = SCP VOLUME USED IN GETTING RMS VALUES !104
! SIGMA_HAT = FLUX COMPONENTS AT PT FROM ORIGINAL ELEMENT !105
! SIGMA_SCP = FLUX COMPONENTS AT PT FROM SMOOTHED SCP !106
! SOLUTION_ERR = DIFFERENCE BETWEEN SOLUTION AND EXACT_SOL !107
! EXACT_SOL = VALUE FROM USER SUPPLIED ROUTINE        !108
! X = COORDINATES OF SYSTEM NODES                     !109
! XYZ = SPACE COORDINATES AT A POINT                   !110

```

Figure 13.3.1b Automatic arrays and local variables


```

WRITE (N_PRT, "(/, '** BEGINNING ELEMENT ERROR ESTIMATES **')") !111
ELEM_ERROR_ENERGY = 0.d0 ; ERR_MAX = 0.d0 ! INITIAL !112
ELEM_REFINEMENT = 0.d0 ; EXAC_ERROR_ENERGY = 0.d0 ! INITIAL !114
NEAR_ZERO = TINY (1.d0) ! CONSTANT !115
!116
GLOBAL_ERROR_ENERGY = 0.d0 ! INITIALIZE !117
GLOBAL_FLUX_NORM = 0.d0 ; GLOBAL_FLUX_ERROR = 0.d0 !118
GLOBAL_FLUX_RMS = 0.d0 ; GLOBAL_SOLUTION_L2 = 0.d0 !119
SCP_VOLUME = 0.d0 ; GLOBAL_SOLUTION_ERR = 0.d0 !120
MEASURE = 0.d0 ; GLOBAL_STRAIN_ENERGY = 0.d0 !121
EXACT_FLUX_ERROR = 0.d0 ; EXACT_FLUX_NORM = 0.d0 !122
EXACT_STRAIN_ENERGY = 0.d0 ; EXACT_ERR_ENERGY = 0.d0 !123
EXACT_SOL = 0.d0 ; EXACT_SOL_L2 = 0.d0 !124
EXAC_ERROR_ENERGY = 0.d0 ; DIFF = 0.d0 !125
GLOBAL_H1_NORM = 0.d0 ; GLOBAL_H2_NORM = 0.d0 !126
GLOBAL_H1_ERROR = 0.d0 ; GLOBAL_H2_ERROR = 0.d0 !127
ELEM_H1_NORM = 0.d0 ; ELEM_H2_NORM = 0.d0 !128
ELEM_H1_ERROR = 0.d0 ; ELEM_H2_ERROR = 0.d0 !129
EXACT_H1_NORM = 0.d0 ; EXACT_H2_NORM = 0.d0 !130
EXACT_H1_ERROR = 0.d0 ; EXACT_H2_ERROR = 0.d0 !131
XYZ = 0.d0 ; IE = 1 !132
!133
! CHECK FOR POSSIBLE CONSTANT CONSTITUTIVE MATRIX !134
IF ( L_HOMO == 1 ) THEN ! HOMOGENEOUS !135
  XYZ = 0.d0 ; IE = 1 ! DUMMY ARGUMENTS !136
  IF ( .NOT. GRAD_BASE_ERROR ) CALL & !137
    SELECT_APPLICATION_E_MATRIX (IE, XYZ, E) !138
  CALL INV_SMALL_MAT (N_R_B, E, E_INVERSE, I_ERROR) !139
END IF ! HOMOGENEOUS MATERIAL !140
!141
LT = 1 ; LAST_LT = 0 !142
DO IE = 1, N_ELEMS ! LOOP OVER ALL STANDARD ELEMENTS !143
!--> GET ELEMENT TYPE NUMBER !144
  IF ( N_L_TYPE > 1 ) LT = L_TYPE (IE) ! SAME AS LAST TYPE ? !145
  IF ( LT /= LAST_LT ) THEN ! this is a new type !146
    CALL GET_ELEM_TYPE_DATA (LT) ! CONTROLS FOR THIS TYPE !147
    LAST_LT = LT !148
  !149
!--> GET UPGRADED QUADRATURE RULE FOR PATCH "ELEMENT" TYPE !150
  CALL GET_PATCH_QUADRATURE_ORDER (LT_SHAP, LT_QP, SCP_QP) !151
!152
! SINCE SCP_QP >= LT_QP MUST REALLOCATE SOME ARRAYS !153
  QP_LT = LT_QP ! Copy to prevent overwrite !154
  LT_QP = SCP_QP ! Return to original value below !155
  IF ( TYPE_APLY_ALLOC ) CALL DEALLOCATE_TYPE_APPLICATION !156
  CALL ALLOCATE_TYPE_APPLICATION !157
  IF ( TYPE_NTRP_ALLOC ) CALL DEALLOCATE_TYPE_INTERPOLATIONS !158
  CALL ALLOCATE_TYPE_INTERPOLATIONS !159
  IF ( ALLOCATED (DOF_EL) ) DEALLOCATE (DOF_EL) !160
  ALLOCATE (DOF_EL (N_G_DOF, LT_N)) !161
!162
  IF ( LT_QP > 0 ) THEN ! GET QUADRATURE FOR PATCH "ELEMENT" !163
    CALL GET_ELEM_QUADRATURES !164
    CALL FILL_TYPE_INTERPOLATIONS ; END IF !165
  END IF ! a new element type !166
!167

```

Figure 13.3.1c Set patch type and quadrature data

```

!-->   GET ELEMENT NODE NUMBERS, COORD, DOF                !168
      ELEM_NODES = GET_ELEM_NODES (IE, LT_N, NODES)         !169
      CALL ELEM_COORD (LT_N, N_SPACE, X, COORD, ELEM_NODES) !170
      INDEX      = GET_ELEM_INDEX (LT_N, ELEM_NODES)       !171
      D          = GET_ELEM_DOF (DOF_SYS)                  !172
      DOF_EL     = RESHAPE (D, (/ N_G_DOF, LT_N /))        !173
                                                    !174
!-->   EXTRACT SCP NODAL FLUXES (NOW GATHER_LT_SCP_AVERAGES) !175
      DO IN = 1, LT_N ! OVER NODES OF ELEMENT              !176
        IF ( ELEM_NODES (IN) < 1 ) CYCLE ! TO VALID NODE  !177
          FLUX_LT (1:N_R_B, IN) = SCP_AVERAGES (           !178
                                &
                                ELEM_NODES (IN), 1:N_R_B) !179
        END DO ! FOR NODES ON ELEMENT                      !180
                                                    !181
!      INITIALIZE NORMS                                     !182
      ELEM_FLUX_NORM      = 0.d0 ; ELEM_FLUX_ERROR        = 0.d0 !183
      ELEM_FLUX_RMS      = 0.d0 ; ELEM_SOLUTION_L2       = 0.d0 !184
      ELEM_SOLUTION_ERR  = 0.d0 ; ELEM_STRAIN_ENERGY     = 0.d0 !185
      EL_ERR_ENERGY     = 0.d0 ; EX_ERR_ENERGY          = 0.d0 !186
      VOL                = 0.d0 ; ELEM_ERROR_ENERGY (IE) = 0.d0 !187
                                                    !188
      DO IQ = 1, LT_QP ! LOOP OVER QUADRATURE POINTS      !189
        H = GET_H_AT_QP (IQ) ! INTERPOLATION FUNCTIONS    !190
        XYZ = MATMUL (H, COORD) ! COORDINATES OF PT      !191
        DLH = GET_DLH_AT_QP (IQ) ! LOCAL DERIVATIVES     !192
                                                    !193
!      FIND JACOBIAN AT THE PT, INVERSE AND DETERMINANT  !194
        AJ = MATMUL (DLH (1:N_SPACE, :), COORD)          !195
        CALL INVERT_JACOBIAN (AJ, AJ_INV, DET, N_SPACE)  !196
        IF ( DET <= ZERO ) STOP 'BAD DET, SCP_ERROR_ESTIMATES' !197
        DET_WT = DET * WT(IQ)                            !198
                                                    !199
        IF ( AXISYMMETRIC ) DET_WT = DET_WT * XYZ (1) * TWO_PI !200
        VOL = VOL + DET_WT ! UPDATE ELEMENT VOLUME       !201
                                                    !202
!      EVALUATE SOLUTION L2 NORM (ASSUMING C_0N_G_DOF)   !203
        SOLUTION      = MATMUL (DOF_EL, H)              !204
        ELEM_SOLUTION_L2 = ELEM_SOLUTION_L2 + DET_WT    & !205
                            * DOT_PRODUCT (SOLUTION, SOLUTION) !206
                                                    !207
!      EVALUATE APPLICATION EXACT VALUE & ERROR HERE    !208
        IF ( USE_EXACT ) THEN                            !209
          CALL SELECT_EXACT_SOLUTION (XYZ, EXACT_SOL)    !210
          EXACT_SOL_L2      = EXACT_SOL_L2 + DET_WT      & !211
                              * DOT_PRODUCT (EXACT_SOL, EXACT_SOL) !212
          SOLUTION_ERR     = EXACT_SOL - SOLUTION        !213
          ELEM_SOLUTION_ERR = ELEM_SOLUTION_ERR + DET_WT & !214
                              * DOT_PRODUCT (SOLUTION_ERR, SOLUTION_ERR) !215
        END IF ! EXACT SOLUTION GIVEN                   !216
                                                    !217
        DGH = MATMUL (AJ_INV, DLH) ! GLOBAL DERIVATIVES !218
        CALL SELECT_APPLICATION_B_MATRIX (DGH, XYZ, B (:,1:LT_FREE)) !219
                                                    !220

```

Figure 13.3.1d Gather continuous flux and integrate error

```

!      GET LOCAL STRAINS (STORE IN SIGMA_HAT)                                !221
      SIGMA_HAT (1:N_R_B) = MATMUL(B(:,1:LT_FREE),D(1:LT_FREE))              !222
                                                                              !223
!      APPLY CONSTITUTIVE RELATION (TO STRAINS IN SIGMA_SCP)                !224
      SIGMA_HAT (1:N_R_B) = MATMUL (E, SIGMA_HAT (1:N_R_B))                  !225
                                                                              !226
!      GET SCP FLUX ESTIMATES & DIFFERENCE                                  !227
      SIGMA_SCP (:) = MATMUL (H, TRANSPOSE(FLUX_LT (:,1:LT_N)))              !228
      DIFF          = SIGMA_SCP - SIGMA_HAT      ! SIGMA ERROR EST           !229
      ELEM_FLUX_NORM = ELEM_FLUX_NORM           &                            !230
                    + DET_WT * DOT_PRODUCT (SIGMA_SCP,SIGMA_SCP)           !231
      ELEM_FLUX_ERROR = ELEM_FLUX_ERROR &                                     !232
                    + DET_WT * DOT_PRODUCT (DIFF, DIFF)                    !233
                                                                              !234
!      INCREMENT STRAIN ENERGY & ENERGY IN THE ERROR                    !235
      TEST = DOT_PRODUCT (SIGMA_SCP,MATMUL(E_INVERSE,SIGMA_SCP))            !236
      ELEM_STRAIN_ENERGY = ELEM_STRAIN_ENERGY + DET_WT * TEST               !237
                                                                              !238
      TEST = DOT_PRODUCT (DIFF, MATMUL (E_INVERSE, DIFF))                   !239
      EL_ERR_ENERGY = EL_ERR_ENERGY + DET_WT * TEST                         !240
      IF (EL_ERR_ENERGY < NEAR_ZERO ) EL_ERR_ENERGY = 0.d0                 !241
                                                                              !242
      IF ( USE_EXACT_FLUX ) THEN ! GET EXACT VALUES                        !243
        CALL SELECT_EXACT_FLUX (XYZ, SIGMA_SCP (1:N_R_B))                   !244
        DIFF          = SIGMA_SCP - SIGMA_HAT ! EXACT ERROR                 !245
        EXACT_FLUX_NORM = EXACT_FLUX_NORM           &                            !246
                    + DET_WT * DOT_PRODUCT (SIGMA_SCP, SIGMA_SCP)           !247
        EXACT_FLUX_ERROR = EXACT_FLUX_ERROR &                                 !248
                    + DET_WT * DOT_PRODUCT (DIFF, DIFF)                    !249
        TEST = DOT_PRODUCT(SIGMA_SCP,MATMUL(E_INVERSE,SIGMA_SCP))          !250
        EXACT_STRAIN_ENERGY = EXACT_STRAIN_ENERGY + DET_WT*TEST            !251
        TEST = DOT_PRODUCT (DIFF, MATMUL (E_INVERSE, DIFF))                !252
        EX_ERR_ENERGY      = EX_ERR_ENERGY      + DET_WT * TEST            !253
        EXACT_ERR_ENERGY = EXACT_ERR_ENERGY + DET_WT * TEST                !254
      END IF ! EXACT FLUXES GIVEN                                           !255
    END DO ! OVER ERROR EST QP                                              !256
                                                                              !257
      EXAC_ERROR_ENERGY (IE) = EX_ERR_ENERGY + NEAR_ZERO                    !258
      ELEM_ERROR_ENERGY (IE) = EL_ERR_ENERGY + NEAR_ZERO                    !259
      MEASURE (IE)          = VOL                                           !260
      SCP_VOLUME           = SCP_VOLUME + VOL                                !261
                                                                              !262
!      COMBINE AND NORMALIZE ERROR TERMS                                    !263
      GLOBAL_STRAIN_ENERGY=GLOBAL_STRAIN_ENERGY+ELEM_STRAIN_ENERGY         !264
      GLOBAL_ERROR_ENERGY = GLOBAL_ERROR_ENERGY + EL_ERR_ENERGY            !265
      GLOBAL_FLUX_NORM    = GLOBAL_FLUX_NORM    + ELEM_FLUX_NORM           !266
      GLOBAL_FLUX_ERROR   = GLOBAL_FLUX_ERROR   + ELEM_FLUX_ERROR          !267
      GLOBAL_SOLUTION_L2  = GLOBAL_SOLUTION_L2  + ELEM_SOLUTION_L2         !268
      GLOBAL_SOLUTION_ERR = GLOBAL_SOLUTION_ERR +ELEM_SOLUTION_ERR         !269
                                                                              !270
      GLOBAL_H2_NORM      = GLOBAL_H2_NORM      + ELEM_H2_NORM             !271
      GLOBAL_H2_ERROR     = GLOBAL_H2_ERROR     + ELEM_H2_ERROR            !272
                                                                              !273
      ELEM_ERROR_ENERGY (IE) = SQRT (ELEM_ERROR_ENERGY (IE))               !274
      EXAC_ERROR_ENERGY (IE) = SQRT (EXAC_ERROR_ENERGY (IE))               !275
    END DO ! OVER ALL ELEMENTS                                             !276
    LT_QP = QP_LT ! RESET LT_QP TO ITS TRUE VALUE                          !277
                                                                              !278

```

Figure 13.3.1e Update various error measure choices

```

!          FINAL GLOBAL COMBINATIONS                                !279
GLOBAL_STRAIN_ENERGY=GLOBAL_STRAIN_ENERGY+GLOBAL_ERROR_ENERGY !280
!281
EXACT_H2_NORM   = EXACT_H2_NORM   + EXACT_FLUX_ERROR &         !282
                  + EXACT_SOL_L2                                     !283
GLOBAL_H2_NORM  = GLOBAL_H2_NORM  + GLOBAL_FLUX_NORM &         !284
                  + GLOBAL_SOLUTION_L2                             !285
!286
STRAIN_ENERGY_NORM = SQRT (GLOBAL_STRAIN_ENERGY)                !287
ALLOWED_ERROR      = STRAIN_ENERGY_NORM*(PERCENT_ERR_MAX/100) !288
ALLOWED_ERR_DENSITY= ALLOWED_ERROR / SQRT (SCP_VOLUME)         !289
ALLOWED_ERR_PER_EL = ALLOWED_ERROR / SQRT (FLOAT(N_ELEMS))    !290
!291
!          AVOID DIVISION BY ZERO IF THE ERROR IS ZERO          !292
ALLOWED_ERROR      = ALLOWED_ERROR      + NEAR_ZERO            !293
ALLOWED_ERR_DENSITY = ALLOWED_ERR_DENSITY + NEAR_ZERO         !294
ALLOWED_ERR_PER_EL  = ALLOWED_ERR_PER_EL  + NEAR_ZERO         !295
ERR_MAX             = ALLOWED_ERROR       !296
EXACT_FLUX_ERROR    = EXACT_FLUX_ERROR    + NEAR_ZERO         !297
EXACT_H1_ERROR      = EXACT_H1_ERROR      + NEAR_ZERO         !298
GLOBAL_ERROR_ENERGY = GLOBAL_ERROR_ENERGY + NEAR_ZERO         !299
GLOBAL_FLUX_ERROR    = GLOBAL_FLUX_ERROR    + NEAR_ZERO         !300
GLOBAL_H1_ERROR      = GLOBAL_H1_ERROR      + NEAR_ZERO         !301
GLOBAL_SOLUTION_ERR  = GLOBAL_SOLUTION_ERR  + NEAR_ZERO         !302
!303
GLOBAL_ERROR_ENERGY = SQRT (GLOBAL_ERROR_ENERGY)                !304
GLOBAL_FLUX_NORM    = SQRT (GLOBAL_FLUX_NORM)                    !305
GLOBAL_SOLUTION_L2  = SQRT (GLOBAL_SOLUTION_L2)                  !306
EXACT_SOL_L2        = SQRT (EXACT_SOL_L2)                        !307
GLOBAL_SOLUTION_ERR = SQRT (GLOBAL_SOLUTION_ERR)                  !308
IF ( SCP_VOLUME > 0.d0 ) GLOBAL_FLUX_RMS = &                   !309
    SQRT (GLOBAL_FLUX_ERROR / SCP_VOLUME)                       !310
GLOBAL_FLUX_ERROR    = SQRT (GLOBAL_FLUX_ERROR)                  !311
!312
!          GET EXACT VALUES, WHEN AVAILABLE                      !313
EXACT_STRAIN_ENERGY = EXACT_STRAIN_ENERGY+EXACT_ERR_ENERGY    !314
EXACT_STRAIN_ENERGY = SQRT (EXACT_STRAIN_ENERGY)                !315
EXACT_FLUX_NORM     = SQRT (EXACT_FLUX_NORM)                    !316
EXACT_FLUX_ERROR    = SQRT (EXACT_FLUX_ERROR)                   !317
!318
EXACT_H2_NORM       = SQRT (EXACT_H2_NORM)                       !319
GLOBAL_H2_NORM      = SQRT (GLOBAL_H2_NORM)                       !320
EXACT_H2_ERROR      = SQRT (EXACT_H2_ERROR)                       !321
GLOBAL_H2_ERROR     = SQRT (GLOBAL_H2_ERROR)                       !322
!323

```

Figure 13.3.1f Update global error measures

```

PRINT *, "*** S_C_P ENERGY NORM ERROR ESTIMATE DATA ***"      !324
PRINT *, " "                                                       !325
PRINT *, "DOMAIN MEASURE .....", SCP_VOLUME                       !326
PRINT *, "AVERAGE ELEMENT MEASURE ...", SCP_VOLUME / N_ELEMS     !327
PRINT *, "GLOBAL_SOLUTION_L2 .....", GLOBAL_SOLUTION_L2          !328
IF ( USE_EXACT ) THEN                                             !329
  PRINT *, "EXACT_SOLUTION_L2 .....", EXACT_SOL_L2                !330
  PRINT *, "GLOBAL_SOLUTION_ERR.....", GLOBAL_SOLUTION_ERR        !331
END IF ! EXACT SOLUTION GIVEN                                     !332
PRINT *, " "                                                       !333
PRINT *, "STRAIN_ENERGY_NORM .....", STRAIN_ENERGY_NORM          !334
IF ( USE_EXACT_FLUX ) PRINT *,                                     & !335
  "EXACT_STRAIN_ENERGY_NORM ..", EXACT_STRAIN_ENERGY             !336
PRINT *, "ALLOWED_PER_CENT_ERROR ....", PERCENT_ERR_MAX           !337
PRINT *, "ALLOWED_GLOBAL_ERROR .....", ALLOWED_ERROR             !338
PRINT *, "ALLOWED_ERROR_DENSITY ....", ALLOWED_ERR_DENSITY        !339
PRINT *, "ALLOWED_ERROR_PER_ELEM ....", ALLOWED_ERR_PER_EL       !340
PRINT *, " "                                                       !341
PRINT *, "GLOBAL_ERROR_ENERGY .....", GLOBAL_ERROR_ENERGY        !342
PRINT *, "GLOBAL_ERROR_PARAMETER ....", &                         !343
  GLOBAL_ERROR_ENERGY / ALLOWED_ERROR                             !344
PRINT *, " "                                                       !345
PRINT *, "GLOBAL_FLUX_ERROR .....", GLOBAL_FLUX_ERROR            !346
IF ( USE_EXACT_FLUX ) PRINT *,                                     & !347
  "EXACT_FLUX_ERROR .....", EXACT_FLUX_ERROR                    !348
PRINT *, "GLOBAL_FLUX_NORM .....", GLOBAL_FLUX_NORM              !349
IF ( USE_EXACT_FLUX ) PRINT *,                                     & !350
  "EXACT_FLUX_NORM .....", EXACT_FLUX_NORM                      !351
PRINT *, "GLOBAL_FLUX_RMS .....", GLOBAL_FLUX_RMS                !352
                                                                    !353
!      CONVERT TO ELEMENT ERROR DENSITY                           !354
WHERE ( MEASURE > 0.d0 )                                          !355
  ELEM_ERROR_DENSITY = ELEM_ERROR_ENERGY / SQRT (MEASURE)        !356
ELSEWHERE                                                         !357
  ELEM_ERROR_DENSITY = 0.d0                                       !358
END WHERE                                                         !359
                                                                    !360
!      LIST AVERAGE TOTAL ERROR                                  !361
TEMP = STRAIN_ENERGY_NORM / 100.d0                                !362
TEST = SUM ( ELEM_ERROR_ENERGY ) / TEMP                           !363
WRITE(N_PRT, '( "TOTAL % ERROR IN ENERGY NORM = ",1PE8.2)') TEST !364
                                                                    !365
!      LIST MAXIMUMS                                           !366
TEST = MAXVAL ( ELEM_ERROR_ENERGY )                               !367
LOC_MAX = MAXLOC ( ELEM_ERROR_ENERGY )                             !368
PRINT *, " "                                                       !369
WRITE (N_PRT, '( "MAX ELEMENT ENERGY ERROR OF ",1PE8.2)') TEST !370
WRITE (N_PRT, '( "OCCURS IN ELEMENT ", I6)') LOC_MAX (1)         !371
TEST = MAXVAL ( ELEM_ERROR_DENSITY )                               !372
LOC_MAX = MAXLOC ( ELEM_ERROR_DENSITY )                             !373
WRITE (N_PRT, '( "MAX ENERGY ERROR DENSITY OF ",1PE8.2)') TEST !374
WRITE (N_PRT, '( "OCCURS IN ELEMENT ", I6)') LOC_MAX (1)         !375
                                                                    !376

```

Figure 13.3.1g List global error measures

```

!   FINALLY, CONVERT REFINEMENT TO TRUE REFINEMENT PARAMETER      !377
ELEM_REFINEMENT = ELEM_ERROR_DENSITY * SQRT ( SCP_VOLUME ) & !378
                  / ALLOWED_ERROR                                !379
WRITE (N_PRT, '( "WITH REFINEMENT PARAMETER OF ", 1PE8.2)') & !380
      TEST / ALLOWED_ERR_DENSITY                                !381
PRINT *, "-----"                                           !382
PRINT *, "          ERROR IN          % ERROR IN      REFINEMENT" !383
PRINT *, "ELEMENT,  ENERGY_NORM,  ENERGY_NORM,  PARAMETER" !384
PRINT *, "-----"                                           !385
                                                                !386
DO IE = 1, N_ELEMS ! LOOP OVER ALL ELEMENTS                    !387
  IF ( ELEM_ERROR_ENERGY (IE) > ALLOWED_ERR_PER_EL .OR. & !388
      ELEM_ERROR_DENSITY (IE) > ALLOWED_ERR_DENSITY ) THEN & !389
    WRITE (N_PRT, "(I8, 3(1PE16.4),6X,A)") IE, & !390
      ELEM_ERROR_ENERGY (IE), ELEM_ERROR_ENERGY (IE) / & !391
      TEMP, ELEM_REFINEMENT (IE), "Refine"                !392
  ELSE                                                         !393
    WRITE (N_PRT, "(I8, 3(1PE16.4),6X,A)") IE, & !394
      ELEM_ERROR_ENERGY (IE), ELEM_ERROR_ENERGY (IE) / & !395
      TEMP, ELEM_REFINEMENT (IE), "Refine" ; END IF        !396
END DO ! FOR ALL ELEMENTS                                     !397
                                                                !398
!   CONVERT TO % ERROR IN ENERGY NORM * 100                  !399
ELEM_ERROR_ENERGY = ELEM_ERROR_ENERGY / TEMP                !400
IF ( TYPE_APLY_ALLOC ) CALL DEALLOCATE_TYPE_APPLICATION    !401
IF ( TYPE_NTRP_ALLOC ) CALL DEALLOCATE_TYPE_INTERPOLATIONS !402
END SUBROUTINE SCP_ERROR_ESTIMATES                          !403

```

Figure 13.3.1f List element error and error density

$$\begin{pmatrix} \frac{\partial^2}{\partial r^2} \\ \frac{\partial^2}{\partial s^2} \\ \frac{\partial^2}{\partial r \partial s} \end{pmatrix} = \begin{pmatrix} \frac{\partial^2 x}{\partial r^2} & \frac{\partial^2 y}{\partial r^2} \\ \frac{\partial^2 x}{\partial s^2} & \frac{\partial^2 y}{\partial s^2} \\ \frac{\partial^2 x}{\partial r \partial s} & \frac{\partial^2 y}{\partial r \partial s} \end{pmatrix} \begin{pmatrix} \frac{\partial}{\partial x} \\ \frac{\partial}{\partial y} \end{pmatrix} + \begin{pmatrix} \left(\frac{\partial x}{\partial r}\right)^2 & \left(\frac{\partial y}{\partial r}\right)^2 & 2 \frac{\partial x}{\partial r} \frac{\partial y}{\partial r} \\ \left(\frac{\partial x}{\partial s}\right)^2 & \left(\frac{\partial y}{\partial s}\right)^2 & 2 \frac{\partial x}{\partial s} \frac{\partial y}{\partial s} \\ \frac{\partial x}{\partial r} \frac{\partial x}{\partial s} & \frac{\partial y}{\partial r} \frac{\partial y}{\partial s} & \frac{\partial x}{\partial s} \frac{\partial y}{\partial r} + \frac{\partial x}{\partial r} \frac{\partial y}{\partial s} \end{pmatrix} \begin{pmatrix} \frac{\partial^2}{\partial x^2} \\ \frac{\partial^2}{\partial y^2} \\ \frac{\partial^2}{\partial x \partial y} \end{pmatrix}. \quad (13.3)$$

For a constant Jacobian the first rectangular matrix on the right is zero. Otherwise, the second derivatives are clearly more sensitive to a variable Jacobian. In each case, we must invert the square matrices to obtain the first and second physical derivatives. If the Jacobian is not constant, then the effect of a distorted element would be amplified by the product terms in the square matrix of Eq. (13.3), as well as by the second derivatives in the rectangular array that multiplies the physical gradient term. Using this analytic form for the second derivative of an approximate finite element solution is certainly questionable. Clearly, for a linear interpolation element with a constant Jacobian, the result would be identically zero. Thus, we will actually only use this form as a tool to estimate regions of high second-derivative error, as compared to the values obtained from a patch recovery technique.

In the present software the second derivative calculations and associated output are activated by a global logical constant `SCP_2ND_DERIV` which is set to true by including an input keyword control of `scp_2nd_deriv` in the data file. If the norm of the estimated

second derivative error is required, then the square matrix of Jacobian product terms is computed in subroutine JACOBIAN_PRODUCTS and is given the name P_AJ. Its associated inverse matrix is called P_AJ_INV. The second parametric (local) derivatives of the interpolation functions, \mathbf{H} , are denoted as $\mathbf{D2LH}$ and the corresponding second physical (global) derivatives, from Eq. (13.3), are denoted by $\mathbf{D2GH}$. Each has N_2_DER rows and a column for each interpolation function. For one-, two- and three-dimensional problems, N_2_DER has a value of 1, 3, or 6, respectively.

Since the analytic estimate will usually be compared to a smoothed patch estimate, the second derivative terms are also computed in EVAL_SCP_FIT_AT_PATCH_NODES if SCP_2ND_DERIV is true. After the element fluxes have been processed to give continuous nodal values on a patch (as described earlier), the physical gradients of the patch interpolations, SCP_DGH, are invoked to evaluate the gradients of the fluxes (i.e., the second derivatives) at all of the mesh nodes in the patch. They are then also scattered to the system array, SCP_AVERAGES, for later averaging over all patch contributions. Sometimes one may want to bias estimates on the boundary of the domain before scattering its contribution to the system averages.

After the flux components and their gradients (second derivatives) have been averaged, they are listed at the nodes, and/or saved for plotting and then passed to the routine SCP_ERROR_ESTIMATES. There the second derivative values are only used to calculate various measures or norms for them and their estimated error. While the corresponding cross derivatives like $\partial^2/\partial x\partial y$ and $\partial^2/\partial y\partial x$ should be equal, they generally will differ due to various numerical approximations. All cross derivatives are computed, but only average values are used in estimating errors in the second derivatives. The larger full set of second derivatives at the nodes of an element are gathered and placed in an array called DERIV2_LT. If cross derivative estimates exist, they are averaged and placed in a smaller array called DERIV2_AVE that has the N_2_DER second derivatives at each element node. They will be interpolated to give the average second derivatives at the quadrature points used to evaluate the norms. By analogy to the first derivative norms, the N_2_DER interpolated second derivatives at a point are called DERIV2_SCP. The values computed directly from Eq. (13.3) are called DERIV2_HAT. Their values and differences are used to define norms and error estimates at the element level (ELEM_H2_NORM and ELEM_H2_ERROR) and at the system level (GLOBAL_H2_NORM and GLOBAL_H2_ERROR). If an exact solution is available for comparison, called FLUX_GRAD, it is used to compute corresponding exact values of the second spatial derivatives (EXACT_H2_NORM and EXACT_H2_ERROR). The various norm and error measures are listed, as are the SCP averaged and exact second derivatives at the system nodes. For plotting or other use, the last two items are saved to external files called pt_ave_grad_flux.tmp and pt_ex_grad_flux.tmp, respectively.

13.7 References

- [1] Ainsworth, M. and Oden, J.T., *A Posteriori Error Estimation in Finite Element Analysis*, New York: John Wiley (2000).
- [2] Akin, J.E. and Maddox, J.R., "A RP-Adaptive Scheme for the Finite Element Analysis of Linear Elliptic Problems," pp. 427–438 in *The Mathematics of Finite Elements and Applications*, ed. J.R. Whiteman, London: Academic Press (1996).
- [3] Blacker, T. and Belytschko, T., "Superconvergent Patch Recovery with Equilibrium and Conjoint Interpolant Enhancements," *Int. J. Num. Meth. Eng.*, **37**, pp. 517–536 (1995).
- [4] Cook, R.D., Malkus, D.S., Plesha, N.E., and Witt, R.J., *Concepts and Applications of Finite Element Analysis*, New York: John Wiley (2002).
- [5] Krizek, M., Neittaanmaki, P., and Stenberg, R., *Finite Element Methods: Superconvergence, Post-Processing and a Posteriori Estimates*, New York: Marcel Dekker, Inc. (1998).
- [6] Lakhany, A.M. and Whiteman, J.R., "Superconvergent Recovery Operators," pp. 195–215 in *Finite Element Methods: Superconvergence, Post-Processing and a Posteriori Estimates*, New York: Marcel Dekker, Inc. (1998).
- [7] Loubignac, G., Cantin, G., and Touzot, G., "Continuous Stress Fields in Finite Element Analysis," *AIAA Journal*, **15**(11), pp. 239–241 (1977).
- [8] Press, W.H., Teukolsky, S.A., Vetterling, W.T., and Flannery, B.P., *Numerical Recipes in Fortran 90*, New York: Cambridge University Press (1996).
- [9] Szabo, B. and Babuska, I., *Finite Element Analysis*, New York: John Wiley (1991).
- [10] Wiberg, N-E., Abdulwahab, F., and Ziukas, S., "Enhanced Superconvergent Patch Recovery Incorporating Equilibrium and Boundary Conditions," *Int. J. Num. Meth. Eng.*, **37**, pp. 3417–3440 (1994).
- [11] Wiberg, N-E., Abdulwahab, F., and Ziukas, S., "Improved Element Stresses for Node and Element Patches Using Superconvergent Patch Recovery," *Com. Num. Meth. Eng.*, **11**, pp. 619–627 (1995).
- [12] Wiberg, N-E., "Superconvergent Patch Recovery – A Key to Quality Assessed FE Solutions," *Adv. Eng. Software*, **28**, pp. 85–95 (1997).
- [13] Zhu, J.Z. and Zienkiewicz, O.C., "Superconvergence Recovery Techniques and A Posteriori Error Estimators," *Int. J. Num. Meth. Eng.*, **30**, pp. 1321–1339 (1990).
- [14] Zhu, J.Z., "Derivative Recovery Techniques and a Posteriori Error Estimation in the Finite Element," *SIAM J. Appl. Num. Anal.*, **N**, pp. nn–nnn (1992).
- [15] Zienkiewicz, O.C. and Zhu, J.Z., "A Simple Error Estimator and Adaptive Procedure for Practical Engineering Analysis," *Int. J. Num. Meth. Eng.*, **24**, pp. 337–357 (1987).
- [16] Zienkiewicz, O.C. and Zhu, J.Z., "Superconvergent Patch Recovery Techniques and Adaptive Finite Element Refinement," *Comp. Meth. Appl. Mech. Eng.*, **101**, pp. 207–224 (1992).

- [17] Zienkiewicz, O.C. and Zhu, J.Z., "The Superconvergent Patch Recovery and *a Posteriori* Error Estimates. Part 2: Error Estimates and Adaptivity," *Int. J. Num. Meth. Eng.*, **33**, pp. 1365–1382 (1992).
- [18] Zienkiewicz, O.C. and Taylor, R.L., *The Finite Element Method*, 5th Edition, London: Arnold (2000).