Using the StressRefine Library to Add p-adaptivity to a Finite Element Code

Note: whereever functions calls are mentioned below, the functions are defined in SrlibSimple in file globalWrappers.cpp

There are two steps:

1. Setup so element and stress recovery routines work for arbitrary p-order
2. Wrap an adaptive loop around your solution routine

I will illustrate how to carry out these steps by showing modifications to the pseudo-code of a conventional finite element program in red.

Notes on programming languages:

- the StressRefine library was designed to be called directly from C++
- wrappers have been provided so all the needed functions can be also called from c
- wrappers have been provided so all the needed functions can be also called from fortran using linux conventions, for example, for a routine cfun(), a wrapper cfun_() is provided, and the function is call from fortran using "call cfun()". In the descriptions below, I always give the name of the c-function. The corresponding Fortran-calleable functions has a lower case "f" in front. For example, to set the number of nodes in the model, the c function is `SetNumNodes, while from fortran it would be "call fSetNumNodes".`

1. Setup for p-adaptive elements

Here is the pseudo-code for finite element setup and linear solution

- Read mesh information. Nodal Coordinates, Element definition (node ids and materials), material properties, loads and constraints
- element bookkeeping- relate local node number to global node id for each element (this is often called the ID array in textbooks)
- process loads
- process constraints- determine which global degrees of freedom are constrained, and determine global equation numbers for the unconstrained dofs, and assemble enforced displacement vector
- calculate element stiffnesses
- Assemble global stiffness and decomp
- backsolve load vector to get global displacement vector
- Evaluate stress: loop over elements
    - download global displacement to local displacements in element
    - call element stress routine

- Postprocess and output in desired format

Here are the modifications needed to make the elements work for arbitrary p-orders, shown in red

These steps are only done once to set up the problem:

- Read mesh information. Nodal Coordinates, Element definition (node ids and materials), material properties, loads and constraints
- set number of nodes in model: `SetNumNodes(n)`
- input nodal coordinates: for each node: `createNode(int uid, double x, double y, double z)`
- Enter Number of elements of each type (bricks, wedges, tets): `SetNumElements(int numBricks, int numWedges, int numTets)`
    - Note: this must be called even if your are modifying your own elements. stressRefine stores informaton needed for the basis function routines in the elements
- Use element definitions to create edges and faces for the model
    - Note: this must be called even if your are modifying your own elements. stressRefine stores informaton needed for the basis function routines in the elements
    - For each element, `createElementIso(int uid, int numnodes, int* nodeids, double E, double nu);`
    - after all elements have been created, create global faces: `createGlobalFaces()`
- input nodal and face constraints: `allocateConstraints(int ncon).` `Then for each nodal constraint:` `inputNodalConstraint(int nid, bool constraineddof[3], double enforcedDisp[3]);`
  `for each face constraint:` `addFaceConstraint(int elemId, int* nidv, bool constraineddof[3], bool anyenfd);`
  `This returns the constraintid that was added. If there are enforced displacements, for each corner node of the face,` `inputFaceNodeEnfd(int constraintId, int localNodeNum, double enforcedDisp[3]);`
  `where` `constraintId` `was returned by` `addFaceConstraint`
- If there are any constraints in coordinates systems other than the global coordinate system (gcs), they need to be preprocessed so they can be handled via a penalty method: `PreProcessPenaltyConstraints`

These steps must be repeated for every solution pass:

- element bookkeeping- relate local node number to global node id for each element (this is often called the ID array in textbooks)
    - use the p-orders for the element's edges to determine number of functions and relate element local function numbers to global functions numbers. routines
- process loads
    - No changes are required in processing loads as long as they are nodel, or constant, linear or quadratic distributed surface loads or body loads. They may be processed the same way as for conventional quadratic elements

- process constraints- determine which global degrees of freedom are constrained, and determine global equation numbers for the unconstrained dofs, and assemble enforced displacement vector. This is only for constraints in gcs: `ProcessConstraints`
  (non-gcs constraints are handled by the above call to `PreProcessPenaltyConstraints` and during element stiffness calculation)
- number equations: fill up array functionsequations for each function and degree of freedom in the model: `numberEquations. This returns total number of equations. Equation numbers corresponding to each function, dof can then be accessed using GetFunctionEquation(fun, dof). This returns an equation number if the function, dof is unconstrained, else -1.`
- calculate element stiffnesses
  - either modify your code's element routines to use p-adaptive displacement functions instead of conventional shape functions (discussed below) OR
  - calculate element stiffnesses using stressRefine elements. Since these use stressRefine's internal quadratic mapping, first call `mapSetup, then for each element: CalculateStiffnessMatrix (id, upperTriangle)`
  - NOTES:
    1. SRelement routine will automatically handle the penalty constraints at the element level. If you are using your own stiffness routines they will have to be modified to handle this.
    2. If using the SRelement routine, it is recommended to call `checkElementMapping before entering the loop to calculate elements stiffnesses. This will perform element "partial flattening" link!! if any elements have invalid mapping`
- Assemble global stiffness and decomp
  - use the modification to do the element bookkeeping
  - No effect on direct solvers
  - For iterative solvers, preconditioners that work with conventional elements may not perform as well with higher order elements.
- backsolve load vector to get global displacement vector
- Evaluate stress: loop over elements
  - download global displacement to local displacements in element
  - call element stress routine
    1. either modify your code's element stress routines to use p-adaptive displacement functions instead of conventional shape functions (discussed below) OR
    2. evaluate stresses with stressRefine routine. routine void calculateRawStress(int elemId, double r, double s, double t, double* stress)
       Note: hierarchical basis functions cannot be evaluated directly at corners, they are singular. Either evaluate them away from the corner and project OR evaluate them at optimal sampling points and smooth to corners and midnodes (recommended, more accurate): see post.globalstrainsmooth in SRwithMklProj `This calculates the smoothed strain vector for each function of each element. You will need to modify` globalstrainsmooth if you are using a different solver than Intel Mkl Pardiso.

Then to calculate the stress from the smoothed strains at any point in an element:
void calculateSmoothedStress(int elemId, double r, double s, double t, double* stress);

Postprocess and output in desired format

2. Adaptive Solution loop

- Initialization: perform the steps from “read mesh information…” to “PreProcessPenaltyConstraints”

LOOP:

- Perform the solution steps, starting with “element bookkeeping”, down to “Evaluate stress: loop over elements”
- Calculate errors from solution
    - setup: `setupErrorCheck(bool finalAdapt, double stressMax, double ErrorTolerance, int maxPorder, int maxPJump, int maxPorderLowStress);` This routine calculates the tractions at sample points for each local face of each element
    - for each element: `FindElementError(int elId);` This returns the element error relative to the max stress in model
- Calcuate new require p-orders: for each element: `FindNextP(int elId, int* pNext);` returns true if the element needed to be adapted else false.
- if no element was adapted, break loop

Until Converged

Modifying Element Stiffness Routines for P-adaptivity

(a similar modification is done for stress Routines)

Wherever your element routine calculates derivatives of the basis functions for displacement, this needs to be replaced with the routines from the stressRefine library: int ElementBasisDerivs(int elemId, double r, double s, double t, double* dbasisdr, double* dbasisds, double* dbasisdt)