FEM_ORDER2 manual

TABLE OF CONTENTS

Overview

The current program is designed to solve the problem of heat transfer and heat-related elastic problem.

The method is traditional FEM performed on a 2nd-order mesh, with nodes in edges but not on in faces or volumes. The mesh is read into the program SEQUENTIALLY, where only process is responsible for that. And all the definitions of the problem, including boundary conditions an constitutional relations are also only given to process 0.

After reading mesh and problem definitions, mesh info is partitioned and distributed to all processes, both topology and needed point coordinates. Meanwhile, the size of needed CSR is counted and distributed so that preallocation for PETSC matrix can be done. Then, to solve the FEM problems, the program conceptually performs integration, both in the volumes and faces, to obtain the stiffness matrix and right hand side vector (RHS). The integration is performed within the normalized space for each face or volume element. As we are using isoparametric method, the jacobian for transformation between physical space to the element's normalized space can be calculated via some simple linear algebraic techniques. In the end, the program uses PETSC's KSP object to solve the linear system.

In the current implementation, while assembling the matrices and RHS, only volume integration is well parallelized while surface integration on boundary conditions is performed only by process 0. When the boundary conditions are relatively small in number, performance is unchanged. I have future plans of parallization of the boundary integrals.

Numerical Methods

In conventional finite-element method, or a common Galerkin method, the trial functions, being identical with the bases of the discrete solution, are considered to be C^0. Therefore the piecewise defined polynomial bases should maintain C^0 continuity on the interfaces of volumes (take 3-D for example). As a result, it is better to consider the discrete DOFs to be set on the vertices, and the bases are interpolation functions that satisfy a kronecker-delta property over the vertices. For example, when each volume is a tetrahedron with 4 vertices, you can easily transform it linearly into a corner of a cartesian box, and using 3 cartesian axes you can easily define first-order bases functions for each vertex. Generally, the basis functions are defined in a normalized coordinate system as polynomials. As the transformation between the normalized space and the geometric space is generally not a linear mapping (basically as curved elements), generally the same set of basis functions are used to interpolate the mapping. Fundamentally, the actual bases are fractions rather than polynomials, but as flat-faced and near flat-faced elements are the majority, the orders of polynomial-based numerical integral are mostly decided with the situation of a linear spacial mapping.

This program uses serendipity method to infer the correct basis functions in the normal space, which is 2nd order polynomials. The exact formulae can be found in the source code.

Using the interpolation described above, a Galerkin method can be derived. For linear heat transfer and linear elasticity, the Galerkin discretion causes the ODEs:

$$M_t\ddot{a}_t + C_t\dot{a}_t + K_t a_t = F_t$$

$$M\ddot{a} + C\dot{a} + Ka = F$$

Where the subscripts 't' denote that it's related to the heat transfer problem, or elastic problem otherwise. a and a_t are just DOFs or nodal values, in the elastic problems, we denote that for nodes $[i_0, i_1, ...]$, a is column vector $[u_0, v_0, w_0, u_1, v_1, w_1, ...]^T$, where $[u, v, w]^T$ is the displacement of some point.
In common problems, only M, M_t, K, K_t, F, F_t are desired, so we discuss them here first:

Constructing The Linear Elastic Problem

The Galerkin method tells us that:

$$K = \int_\Omega B^T D B dV + \int_{\partial\Omega} N^T H N d\Gamma, \quad M = \int_\Omega N^T \rho N dV,$$

$$F = \int_\Omega f^T N dV + \int_{\partial\Omega} p^T N d\Gamma + \int_{\partial\Omega} (H x_0)^T N d\Gamma$$

Ω is the domain of definition, and dV denotes its differential. $\partial\Omega$ is the boundary of the domain of definition, and $d\Gamma$ denotes its differential.

In the formulae above, B is the function transforming a into the interpolated strain field (using a as the right vector), where $B(x,y,z)_{ij}a_j = e(x,y,z)_i$, and $i = 1, 2, ..., 6$ is the subscript for the components of the strain vector, where the transposing and matrix products are on the i, j subscripts.

$D = D_{ij}$ is the 6x6 constitutional relation transforming strain into stress, where $\sigma_i = D_{ij}e_j$.

N transforms a into the interpolated displacement field, it's definition is similar with B, where $N(x,y,z)_{ij}a_j = disp(x,y,z)_i \equiv [u,v,w]^T(x,y,z)_i$.

H is the 3x3 matrix defining the stiffness of a linearly elastic basement, (not a scalar instead of a matrix for the stiffness could very likely be anisotropic), transforming displacement form base-point x_0 to facial force. H is zero on the boundaries not defined as a elastic basement. H should be symmetric and positive semidefinite to maintain correct physical meaning, that is, it is actually defined by its 3 eigenvalues and 3 eigenvectors.

ρ is the density field, defined as a scalar field.

f is the volume force, including the effect of prestress caused by temperature change.

p is only not zero on known-force boundaries, it is a 3d vector field.

Hx_0 can be viewed as the payload force caused by elastic basement boundaries. Actually, in the program, force boundaries are performed with the elastic base boundary scheme. As the inputs are Hx_0 and H, setting $H = 0$ (while $x_0 \to \infty$, which is not the numeric input) means the boundary condition becomes a given force.

The discrete values of K, M, F are integrated on the piecewise polynomials, where (including N, B) they can actually be decomposed into each element (of volume or face), (for 3d case) we define K_e, M_e, F_e, which are the results of integrals performing only in the volume e. More over, the ordering of K_e, M_e, F_e are solely consistent with the part of a (DOFs on nodes) adjacent to the volume (the other parts are zero in the global matrices and vectors). Therefore, the general procedure of assembling the global matrices is: 1. Calculate the local matrices in local order; 2. Find the map from local order to global order and cumulate the values into the global matrix.

Calculation of the local matrices is essentially a integration within the element. Given a proper numerical integration method, i.e. the locations of the integral points and corresponding weights, the integration is iterating through the integral points, calculating desired value on them and cumulating the values by the weights. The differential quantities dV and $d\Gamma$ are different in the physical space than in the element's normalized space, so

an additional correction with the Jacobian determinant of the space transformation is needed. Also, when considering space derivatives on the integral points, the Jacobian matrix of the space transformation is needed. We give the detailed calculation formula of K_e as below:

$$K_e = \sum_{ip} B_e^T D B_e |J| iw$$
$$B_e = SJ^{-1}Q_e$$

Where B_e, J, Q_e are special values on each integral point and volume, ip denotes the collection of integral points and iw denotes corresponding weight.

$D$is the constitutional matrix mentioned above;

S is the matrix transforming first space derivatives of displacement into strain:

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & \frac{1}{2} & 0 & \frac{1}{2} & 0 \\ 0 & 0 & \frac{1}{2} & 0 & 0 & 0 & \frac{1}{2} & 0 & 0 \\ 0 & \frac{1}{2} & 0 & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

for

$$Sd = \varepsilon$$
$$d = \begin{bmatrix} \frac{\partial u}{\partial x} & \frac{\partial u}{\partial y} & \frac{\partial u}{\partial z} & \frac{\partial v}{\partial x} & \frac{\partial v}{\partial y} & \frac{\partial v}{\partial z} & \frac{\partial w}{\partial x} & \frac{\partial w}{\partial y} & \frac{\partial w}{\partial z} \end{bmatrix}^T \quad \varepsilon = \begin{bmatrix} \varepsilon_{11} \varepsilon_{22} \varepsilon_{33} \varepsilon_{23} \varepsilon_{31} \varepsilon_{12} \end{bmatrix}^T$$

Jacobian J is defined as:

$$J_{ij} = \frac{\partial x_j}{\partial \xi_i}$$

and calculated as

$$J_{ij} = coord_{im} dN_{mj}$$

Where $coord_{im}$ is the ith spacial dimension and mth local node's coordinate value, dN_{mj} is the mth local nodes's shape function's jth spacial derivative (at this integral point, relative to normalized coordinates ξ_i). $coord$ varies with the chosen volume and dN is constant for each kind of volume element (and varies with the integral point).

Matrix Q_e is defined as $Q_{e\,ij} = dN_{ji}$.

In total, apart from information on integration weights and basic relations, to obtain K_e, one must know: 1.the coordinates of nodes and 2.derivatives (to ξ_i) of shape functions on each point. The former one is to be queried

each time entering a new volume, and the latter one is prepared for each type of volume and treated as constants in the assembling routine.

Other matrices and vectors are all treated likewise, the only difference is in the integrand functions. For example, M_e needs N_e, which is 0th derivative of shape functions. They don't need J^{-1} for correction, but the $|J|$ from dV is still needed.

Parts of the matrices and vectors also exist on the face integrals, the Jacobian becomes 2x3 rather than 3x3 matrix. The correction for face integral $d\Gamma$ becomes the magnitude for those two 3-d vectors' cross-product. The cross product is also the normal direction for the interpolated surface.

The matrices and vectors used in the heat transfer problems are obtained in similar ways with different integrands and also different number of DOFs.

Applying Given Displacement (or Temperature) Boundaries

The given force boundaries (given heat flux) or soft basement (given linear surface heat exchange) boundaries are automatically included when calculating the matrices and vectors without modifying those without them. When the elastic basement gives zero stiffness and finite resultant force (finite Hx_0) it becomes a given force boundary; while a infinite H and finite x_0 turns it into a given displacement boundary. The matter is that an infinite number (or very large in the actual program) as stiffness does harm to the condition number of the global stiffness matrix, and potentially causes efficiency problems in the linear solver.

A straightforward way is to simply throw away the fixed DOFs as they are already known, but it causes problems in programming. Adding Lagrangian multipliers is also viable but it also changes the shape of the matrices and data layout. This program applies the simple way of decoupling those DOFs, which means before applying fixed value v to DOF i:

$$K_0 = \begin{bmatrix} & & k_{1i} & & \\ & & k_{2i} & & \\ & & \dots & & \\ k_{i1} & k_{i2} & \dots k_{ii} \dots & k_{in-1} & k_{in} \\ & & \dots & & \\ & & k_{n-1i} & & \\ & & k_{ni} & & \end{bmatrix}$$

$$F_0 = \begin{bmatrix} f_1 \\ f_2 \\ \dots \\ f_i \\ \dots \\ f_{n-1} \\ f_n \end{bmatrix}$$

after:

$$K_1 = \begin{bmatrix} & & & 0 & & & \\ & & & 0 & & & \\ & & & ... & & & \\ 0 & 0 & ... & k_{ii} & ... & 0 & 0 \\ & & & ... & & & \\ & & & 0 & & & \\ & & & 0 & & & \end{bmatrix}$$

and vector:

$$F_0 = \begin{bmatrix} f_1 - vk_{1i} \\ f_2 - vk_{2i} \\ ... \\ vk_{ii} \\ ... \\ f_{i-1} - vk_{n-1i} \\ f_i - vk_{ni} \end{bmatrix}$$

While if concerning mass matrix, similar operations are done and the diagonal element m_{ii} is set to a very small value to dramatically increase the causing eigenvalue to avoid the desired modes.

Module Brief

fem_order2 in fem_order2.f90

This module is the primary container of all the data and functions directly connected to mesh management, case building, solving and output functions.

Partition Data:

Created for initial partitioning of the mesh, primarily created in **SetUpPartition**.

Cell Partition Data:

Created with the help of Partition Data, primarily created in **DoCellPartition**. Used for distributing the cell data.

Partitioned Mesh:

Distributed mesh, stored in CSR. In the columns of the CSR, denoting the index INODE, if INODE is in [0,localDOFs), then the vertex is a local vertex, its global index is INODE+indexLo. If INODE is in [localDOFs,inf), then the vertex is a ghost vertex, its global index is ghostingGlobal(INODE-localDOFs+1). To acquire nodal data on the vertex, the index INODE(0 based) is used in the local array of Vec objects, and the Vec objects should be equipped with ghost points. For example, localCoords is a 3x Vec object equipped with 3x ghost points, that is, all the elements are expanded into blocks in 3, which means INODE indicates (INODE+1)*3-2~(INODE+1)*3 (1 based) data in the local array of localCoords. This procedure is demonstrated in the SetUpThermalPara and SetUpElasticityPara.

Solver and Solution Data:

Solver and solution data, along with boundary-condition definitions.

Vec dofFix<T>Dist is distributed 1x or 3x nodal data, where if the data is not nan, the nodal dof is mathematically fixed to this value.

Scatterers:

Scatterer objects, to redistribute nodal data between: [1] original data numbering on proc0 and [2] partitioned data numbering on all procs.

Created automatically when calling GatherVec1 or GatherVec3

Auxiliary Data:

Data used temporarily but could be used in some occasion afterwards. Need a clean-up procedure.

localAdjacencyNum and ghostAdjacencyNum are merely indicating the row sizes of the parallel matrix to minimize dynamic memory allocation and space wasting.

FEM Fundamentals:

Some tiny data predefined for FEM-related calculation.

subroutine initializeStatus

To initialize status of some auto-created objects.

Call collectively.

subroutine initializeLib

To initialize tiny data of element libs, concerning the shape functions and integration points in a regularized coordinate.

Call collectively.

subroutine initializeConstitution

To initialize tiny data concerning constitutional relations.

Call collectively.

subroutine ReducePoints

To reduce the unused points in mesh data, actually only concerning module globals.

Call on proc0.

subroutine getVolumes

To calculate cell volumes, useful in checking if input cells are bad. Only on proc0.

Call on proc0.

subroutine output_plt_mesh

To write the mesh file, with data in module globals.

Call on proc0.

subroutine output_plt_scalar

To output a scalar field, either cell-centered or nodal, in original numbering.

Call on proc0.

subroutine output_plt_scalar3x

Same as **output_plt_scalar**, only that data is blocked in 3.

Call on proc0.

subroutine write_binary_plt_string

Internally called, to write strings in tecplot scheme.

subroutine SetUpPartition

Creates partition and distributes the data to all procs.

Call collectively.

subroutine DoCellPartition

Internally called, to distribute cell data to processes.

subroutine DoGeomPartition

Internally called, to distribute node coords to processes.

subroutine SetUpThermalBC_BLOCKED

To preprocess the block-defined boundary conditions. Currently only distributes the fixed boundary condition to procs.

Call collectively.

subroutine SetUpThermal_InitializeObjects

To initialize load vector and stiffness matrix objects for thermal, and do preallocation for the matrix. If if_dynamic_ther is true then preallocate the mass matrix, otherwise the mass matrix is omitted.

Call collectively.

subroutine SetUpThermalPara

To assemble (integrate) the stiffness matrix and the load vector. Basically doing volume and surface integration. If if_dynamic_ther is true then assemble the mass matrix, otherwise the mass matrix is omitted.

Call collectively.

subroutine SetUpElasticBC_BLOCKED

Same as **SetUpThermalBC_BLOCKED** but the elastic part.

Call collectively.

subroutine SetUpElasticity_InitializeObjects

Same as **SetUpThermal_InitializeObjects** but the elastic part.

Call collectively.

subroutine SetUpElasticityPara

Same as **SetUpThermalPara** but the elastic part.

Call collectively.

subroutine SolveThermal_Initialize

To initialize and configure solver objects for thermal problem.

Call collectively.

subroutine SolveThermal

To actually solve the problem.

Call collectively.

subroutine SolveElasticity_Initialize

Same as **SolveThermal_Initialize** but the elastic part.

Call collectively.

subroutine SolveElasticity

Same as **SolveThermal** but the elastic part.

Call collectively.

subroutine GetElasticityUGradient

To calculate the gradient of displacement for a elastic solution.

Call collectively.

subroutine GetStrainStress

To calculate the strain and stress using the gradient of displacement for a elastic solution.

Call collectively.

subroutine GatherVec1

To redistribute nodal data between: [1] original data numbering on proc0 and [2] partitioned data numbering on all procs, using the scatterers.

If ifreverse is true, then scatters [1] to [2], else gathers [2] to [1].

Call collectively.

subroutine GatherVec3

The 3-blocked version of **GatherVec3**.

Call collectively.