Progress report

R Adhithya

Solving an eigenvalue problem in MOOSE

The Eigenvalue problem is solved by using slepc (Scalable Library for Eigenvalue Problem Computations) library. A square plate where the corner points are fixed were taken to calculate the eigenfrequencies. GMSH(.msh) and ABAQUS(.inp) mesh files can be used as input in for MOOSE simulations.

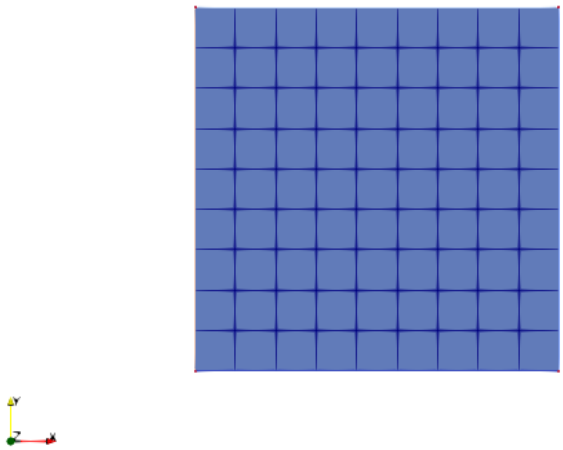

Fig. 1 Mesh file for the square plate problem

The following kernels were used

- MatDiffusion - This kernel is used to solve the term $(\nabla.D, \nabla u)$, where D is a constant
- CoefReaction - This kernel is used to solve the term $(\lambda v, \psi)$. The eigenvalue problem is solved by adding the term *tag = eigen*.

The slepc algorithm references need to be given in the Executioner block. The jacobian free variants (PFJNK, PFJNKMO, JFNK) are designed to give only one eigenvalue. The algorithms like Arnoldi, Krylovschur, jacobi davidson can be used to obtain multiple eigenvalues.

The *Tensor Mechanics* module can be used to incorporate elasticity tensor in the problem instead of a constant diffusivity. The blocks to evaluate the stress divergence term, elasticity tensor and elasticstress needs to be mentioned.

Two input files *eigen2D.i* and *eigen1D_periodic_trial2.i* contain the information about the solving the eigenvalue problem of a 2D square plate and incorporating tensor mechanics module to a 1D problem repsectively. The procedure used is explained but the results were not checked for correctness.

Incorporating Bloch Boundary condition

The bloch boundary condition is defined as follows:

$$\begin{aligned} \mathbf{u}(\mathbf{x}+\mathbf{h_i}) &= \mathbf{u}(\mathbf{x})\exp(i\mathbf{k}.\mathbf{h_i}) \\ \sigma(\mathbf{x}+\mathbf{h_i}) &= \sigma(\mathbf{x})\exp(i\mathbf{k}.\mathbf{h_i}) \end{aligned} \tag{1}$$

The use of complex variables in solving the eigenvalue problem is not incorporated in MOOSE. Therefore, one such approach is found in literature where the bloch boundary condition is implemented by splitting the variables into real and imaginary parts[1].

$$\begin{aligned} \mathbf{u}(\mathbf{x}) &= \mathbf{u}^{\mathbf{Re}}(\mathbf{x}) + i\mathbf{u}^{\mathbf{Im}}(\mathbf{x}) \\ \sigma(\mathbf{x}) &= \sigma^{\mathbf{Re}}(\mathbf{x}) + i\sigma^{\mathbf{Im}}(\mathbf{x}) \end{aligned} \tag{2}$$

Substituting eq. (2) in eq. (1),

$$\begin{aligned} \mathbf{u}^{\mathbf{Re}}(\mathbf{x}+\mathbf{h}) &= \mathbf{u}^{\mathbf{Re}}(\mathbf{x})cos(ka) - \mathbf{u}^{\mathbf{Im}}(\mathbf{x})sin(ka) \\ \mathbf{u}^{\mathbf{Im}}(\mathbf{x}+\mathbf{h}) &= \mathbf{u}^{\mathbf{Re}}(\mathbf{x})sin(ka) + \mathbf{u}^{\mathbf{Im}}(\mathbf{x})cos(ka) \end{aligned} \tag{3}$$

and

$$\begin{aligned} \sigma^{\mathbf{Re}}(\mathbf{x}+\mathbf{h}) &= \sigma^{\mathbf{Im}}(\mathbf{x})sin(ka) - \mathbf{u}^{\mathbf{Re}}(\mathbf{x})cos(ka) \\ \sigma^{\mathbf{Im}}(\mathbf{x}+\mathbf{h}) &= -\sigma^{\mathbf{Re}}(\mathbf{x})sin(ka) - \sigma^{\mathbf{Im}}(\mathbf{x})cos(ka) \end{aligned} \tag{4}$$

Incorprating eq. (3) and eq. (4) in MOOSE was explored. MOOSE functors were used to incorporate eq. (3).

```
ADReal
BlochDirichletBCImag::computeQpValue()
{
  //Get point  locator
  const auto pl = _mesh.getPointLocator();
  //Locate node on the other side of the geometry
  //The translation vector will be a user parameter
  libMesh::Point _translation_vec= libMesh::Point(_lattice_vec, 0.0, 0.0);
  const auto new_node = pl->locate_node(*_current_node - _translation_vec);


  if(!new_node)
    mooseError("Did not find the opposite side value");

  const Moose::NodeArg space_arg = {new_node, Moose::INVALID_BLOCK_ID};

  const Moose::StateArg time_arg = Moose::currentState();

  return _coef(space_arg, time_arg) * (_ur(space_arg, time_arg) *
  sin (_wave_num * _lattice_vec)  - _uim(space_arg, time_arg) *
  cos (_wave_num * _lattice_vec));

}
```

Listing 1 Bloch displacement boundary condition for imaginary

The above code uses MOOSE functors to obtain the values of variables at a node. The *_translation_vec* variable is used to obtain the node at the left side of the boundary i.e.$u(x)$. This lead to the error of adding non-zero entries to the jacobian matrix. The reason was found out later that the dirichletbc class in MOOSE does not allow adding non-zero entries to the jacobian matrix.

The NodalBC class was used to rewrite the code. The jacobian entries were derived for a simple three node 1-D element. The bloch boundary condition equations for displacements are rewritten to derive the jacobian entries.

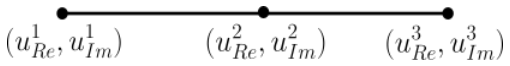

Fig. 2 1-D element used to derive jacobian entries

$$\begin{aligned} u^3_{Re} - u^1_{Re}cos(ka) + u^1_{Im}sin(ka) = 0 \\ u^3_{Im} - u^1_{Re}sin(ka) - u^1_{Im}cos(ka) = 0 \end{aligned} \tag{5}$$

The global jacobian entries can be defined as follows

$$\begin{array}{c} \\ u^1_{Re} \\ u^1_{Im} \\ u^2_{Re} \\ u^2_{Im} \\ u^3_{Re} \\ u^3_{Im} \end{array} \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ -cos(ka) & sin(ka) & 0 & 0 & 1 & 0 \\ -sin(ka) & -cos(ka) & 0 & 0 & 0 & 1 \end{bmatrix} \tag{6}$$

This is implemented using the following code:

```
void
BlochDirichletBCImag::computeOffDiagJacobian(const unsigned int jvar_num)
{
  if (jvar_num == _var.number())
    computeJacobian();
  else
  {
    if (!_var.isNodalDefined())
      return;

    //const Real cached_val = computeQpOffDiagJacobian(jvar_num);

    Real cached_val1 = -cos(_wave_num * _lattice_vec);
    Real cached_val2 = sin(_wave_num * _lattice_vec);

    if (cached_val1 == 0.||cached_val2 == 0)
      // there's no reason to cache this if it's zero, and it can even lead to new nonzero
      // allocations
      return;

    const dof_id_type cached_row = _var.nodalDofIndex();
    // Note: this only works for Lagrange variables...
    const auto pl = _mesh.getPointLocator();
    //Locate node on the other side of the geometry
    //The translation vector will be a user parameter
    libMesh::Point _translation_vec = libMesh::Point(_lattice_vec, 0.0, 0.0);
    const auto _new_node = pl->locate_node(*_current_node - _translation_vec);

    const dof_id_type cached_col = _new_node->dof_number(_sys.number(), jvar_num, 0);

    // Cache the user's computeQpJacobian() value for later use.
    if(cached_col == 0)
    {
      addJacobianElement(_fe_problem.assembly(0, _sys.number()),
                         cached_val1,
                         cached_row,
                         cached_col,
                         /*scaling_factor=*/1);
    }
    else
    {
      addJacobianElement(_fe_problem.assembly(0, _sys.number()),
                  cached_val2,
                  cached_row,
                  cached_col,
                  /*scaling_factor=*/1);
    }
  }
}

}
```

Listing 2 Bloch displacement boundary condition derived using NodalBC class

The entries of the mass matrix has to be edited as well. This is done using the *EigenDirichletBC*. This code gave the error *Zero pivot row 0 value 0*. Upon commenting the *EigenDirichletBC* boundary condition, the code worked. The correctness of the output is being evaluated.

Commercial softwares like ABAQUS use multi-point constraints (MPC) to implement the bloch boundary condition. This was explored by checking whether the constrain system in MOOSE is linked to the EigenSystemBase. Currently, the constraint system is not linked and would require writing a wrapper function to implement this.

Apart from this the XFEM and level set method module were explored to solve the eigenvalue problem for a square plate with a circular hole at the center. Ran into a few errors. The problem will be revisited after successfull implementation of the bloch boundary condition for the 2-D case.

References

[1] Valencia, C., Gomez, J., and Guarín-Zapata, N., "A general-purpose element-based approach to compute dispersion relations in periodic materials with existing finite element codes," *Journal of Theoretical and Computational Acoustics*, Vol. 28, No. 01, 2020, p. 1950005.