FEP Assignment 4

Justin Clough, RIN:661682899

July 27, 2017

1 Introduction

The purpose of this project was to develop a Finite Element Analysis (FEA) code that solves linear elliptic problems. This project does so for two dimensional plane stress solid mechanic problems. The user provides the code with the following information (an example input script is shown in Appendix B.3):

- A geometric model of the domain in the form of a *.dmg* file.
- The corresponding mesh of the domain in the from of a *.smb* file.
- An associations file which define geometric node sets in the form of a *.txt* file. The details of this file are explained in detail in section 3.
- A control file which define material properties, boundary conditions, and linear algebra details in the form of a *.yaml* file.
- A file name to write the solution to as an array of characters.
- The order of numerical integration as an integer.
- The body load in the form of three floating point number.

Based on these input parameters, the code assembles and solves the relevant finite element problem to retrieve the displacement at each mesh Degree of Freedom (DOF). It then calculates the Cauchy stress tensor for each integration point of each element. It finally writes the displacement, traction, and Cauchy stress fields to file in *.vtk* format.

The rest of this report is separated into the following section. The technical description of the code, which includes an overview of the Finite Element Method (FEM), the numerical integration techniques used, and the linear algebra assembly and solution methods is shown in section 2. A description of the code, which includes an outline of classes created and a pseudo code is shown in section 3. The tests performed are described in section 4. Finally, conclusions and closing comments are made in 5. The source code and headers are appended to this report in Appendix A.

2 Technical Description

The purpose of the code is to approximate the displacement u subjected to the conditions shown in Equations 1 through 5:

$$\sigma_{ij},_j - f_i = 0 \quad \text{on} \quad \Omega \tag{1}$$

$$u_i = g_i \quad \text{on} \quad \Gamma_i^g \tag{2}$$

$$\sigma_{ij} \cdot n_j = h_i \quad \text{on} \quad \Gamma_i^h \tag{3}$$

$$\varepsilon_{ij} = u_i,_j \tag{4}$$

$$\sigma_{ij} = c_{ijkm}\varepsilon_{km} \tag{5}$$

where σ is the Cauchy stress tensor in the domain Ω, f is the vector-valued traction, and ε is the strain. The subscripts indicate the spatial dimension of the vector or tensor they follow and range from 1 to n_{sd}, the number of spatial dimensions; summation is implied for repeated indices. The vector n is the outward normal on the pertinent surface The domain Ω is bounded by boundaries Γ^g and Γ^h as shown in Equations 6 through 7:

$$\Omega = \hat{\Omega} \cup \Gamma \tag{6}$$

$$\Gamma = \bigcup_{i=1}^{n_{sd}} \Gamma_i^g \cup \Gamma_i^h \tag{7}$$

where Γ is the total boundary of domain Ω and $\hat{\Omega}$ is the internal portion of Ω. The super scripts, g and h correspond to the Dirichlet and Neumann boundary conditions shown in Equations 2 and 3, respectively. Dirichlet boundary conditions prescribe the vector components of a displacement on a given surface. Neumann boundary conditions prescribe the components of a traction on a given surface. The Dirichlet and Neumann boundary conditions are not defined on the same location for the same spatial dimension, as shown in Equation 8.

$$\Gamma_i^g \cap \Gamma_j^h = \varnothing \quad \text{if} \quad i = j \tag{8}$$

The compliance tensor, c relates the stain and stress by material properties. This application assumed that the material was isotropic and homogeneous. With these assumptions, Equation 5 formed Equation 9:

$$s_i = D_{ij}e_j \tag{9}$$

where s is the Nye-Notation form of σ; e is the Nye-Notation of ε except that the shear strains are doubled. This is shown in Equation 10.

$$\begin{bmatrix} e_1 \\ e_2 \\ e_3 \\ e_4 \\ e_5 \\ e_6 \end{bmatrix} = \begin{bmatrix} \varepsilon_{11} \\ \varepsilon_{22} \\ \varepsilon_{33} \\ 2\varepsilon_{23} \\ 2\varepsilon_{13} \\ 2\varepsilon_{12} \end{bmatrix} \tag{10}$$

The material stiffness tensor D for this application is shown in Equation 11:

$$D = \frac{E}{1-\nu^2} \begin{bmatrix} 1 & \nu & 0 & 0 & 0 & 0 \\ \nu & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & \frac{1-\nu}{2} \end{bmatrix} \tag{11}$$

where E and ν are the Young's Modulus and Poisson Ratio of the material, respectively.

2.1 Finite Element Method

The relations shown in Equations 1 through 5 make up the strong form of the problem. The weak form is shown in Equation 12:

$$\int_\Omega c_{ijkm} w_{i,j}\, u_{k,m}\, d\Omega = \int_\Omega w_i f_i d\Omega + w_i\Big|_{\Gamma_i^h} h_i \tag{12}$$

where w is a weighting function subject to the constraints expressed in Equation 13.

$$w_i \in H^1, \quad w_i\Big|_{\Gamma_i^g} = 0 \quad \forall \quad i = 1(1)n_{sd} \tag{13}$$

In Equation 13, H^1 is the order-one Hilbert space. A derivation from the strong form is shown in Appendix C.1. This weak form can then be approximated by a Galerkin form. The Galerkin form of this problem is shown in Equation 14 and is subject to the constraints and definitions shown in Equations 15 through 18.

$$a(w^h, v^h) = b(w^h, f) + w_i^h\Big|_{\Gamma_i^h} h_i - a(w^h\Big|_{\Gamma_i^g}, g^h) \tag{14}$$

$$a(X,Y) = \int_\Omega c_{ijkm} X_{i,j}\, Y_{k,m}\, d\Omega \tag{15}$$

$$b(X,Y) = \int_\Omega X_i Y_i d\Omega \tag{16}$$

$$u_i^h = v_i^h + g_i^h \tag{17}$$

$$g_i^h \Big|_{\Gamma_i^g} = g_i \qquad v_i^h \Big|_{\Gamma_i^g} = 0 \tag{18}$$

A derivation form the weak from to this Galerkin form is available in Appendix C.2. In Equation 14, u^h is the discrete approximation of u on a discretized Ω, Ω_h. Similarly, w^h is the discrete approximation of w on the same discretized Ω. The discrete approximation of the displacement, u^h, is the sum of unknown, v^h, and given, g^h, displacement values.

As part of the discretization of Ω, shape functions ($N^A(x)$'s) are used to interpolate values. Here, x is a vector describing any position in Ω and has members x_i. Details of the shape functions used discussed in section 2.2. These shape functions are then used shown in in Equations 19 through 21:

$$w_i^h = \sum_{A=1}^{n} c_i^A N^A \tag{19}$$

$$v_i^h = \sum_{A=1}^{n} d_i^A N^A \tag{20}$$

$$g_i^h = \sum_{A=1}^{n} g_i^A N^A \tag{21}$$

where n is the number of discrete evaluation points, or nodes, of Ω_h. The c_i^A's are arbitrary multipliers. From this, a matrix form of the problem can be created which is expressed in Equation 22:

$$a(N^A, N^B) d_i^B = b(N^A, f) + N^A \Big|_{\Gamma_i^h} h_i - a(N^A \Big|_{\Gamma_i^g}, N^C \Big|_{\Gamma_i^g}) g_i^C \tag{22}$$

where A, B, and C represent the set of nodes as described in Equation 23.

$$\begin{aligned} A &\in \eta - \eta_g \\ B &\in \eta - \eta_g \\ C &\in \eta_g \end{aligned} \tag{23}$$

In Equation set 23, η is the set of all nodes on Ω_h and η_g is the subset of η on which Dirichlet boundary conditions are prescribed. Equation 22 can then be rewritten as shown in Equation 24.

$$[K_{AB}]\{d_B\} = \{F_A\} \tag{24}$$

In Equation 24, $[K]$ is the stiffness matrix component relating the degree of freedom of the node at row A to the degree of freedom of the node at column B. The vector $\{d\}$ is comprised of all unknown nodal displacements of the degrees of freedom in Ω_h. The vector $\{F\}$ is comprised of all the nodal force components in Ω_h. A derivation from the Galerkin form shown in Equation 14 to the matrix form shown in Equation 22 is available in Appendix C.3. For situations where Ω_h is discretized with multiple elements, stiffness and force values are summed at each node.

2.2 Numerical Integration

Gaussian quadrature was used to approximate all integrated values in this project. This includes stiffness terms for the elemental stiffness matrices and the elemental force vector components from both traction boundary conditions. A numerical approximation of an integral takes the form shown in Equation 25:

$$\int_{\Omega} p(x)d\Omega = \sum_{n_{int}=1}^{N_{int}} p(x\Big|_{n_{int}})W_{n_{int}} + R \approx \sum_{n_{int}=1}^{N_{int}} p(x\Big|_{n_{int}})W_{n_{int}} \tag{25}$$

where $p(x)$ is the function to integrate, N_{int} is the total number of integration points, $W_{n_{int}}$ is the weight at each integration point, and R is the error in approximating the continuous integral numerically. This error decreases as a higher order of integration is used and was approximated as zero to evaluate calculations. The integration point weights and locations are dependent on the order of approximation and the approximation method. Gaussian quadrature was used for this project. The approximation shown in Equation 25 for three dimensions is shown in Equation 26.

$$\iiint_{\Omega} p(x_1, x_2, x_3)d\Omega \approx \sum_{n_{int}^1=1}^{N_{int}^1} \sum_{n_{int}^2=1}^{N_{int}^2} \sum_{n_{int}^3=1}^{N_{int}^3} p(x_1\Big|_{n_{int}^1}, x_2\Big|_{n_{int}^2}, x_3\Big|_{n_{int}^3})W_{n_{int}^1}W_{n_{int}^2}W_{n_{int}^3} \tag{26}$$

The numerical integration took place on a parametric space, $\square$, that represented each element and used a mapping to return to the domain, Ω_h. The mapping between real and parametric domains is described in Equations 27 through 29.

$$x = (x_1, x_2, x_3) \in \hat{\Omega} \tag{27}$$

$$\xi = (\xi_1, \xi_2, \xi_3) \in \square \tag{28}$$

$$J = det \begin{bmatrix} \frac{\partial x_1}{\partial \xi_1} & \frac{\partial x_1}{\partial \xi_2} & \frac{\partial x_1}{\partial \xi_3} \\ \frac{\partial x_2}{\partial \xi_1} & \frac{\partial x_2}{\partial \xi_2} & \frac{\partial x_2}{\partial \xi_3} \\ \frac{\partial x_3}{\partial \xi_1} & \frac{\partial x_3}{\partial \xi_2} & \frac{\partial x_3}{\partial \xi_3} \end{bmatrix} \tag{29}$$

The variable J is the Jacobian determinant and represents the volumetric dilation from $\square$ to Ω_h for a given location ξ. Combining the mapping shown in Equations 27 through 29 with the three dimensional numerical integration technique shown in Equation 26 yield Equation 30.

$$\iiint_{\Omega} p(x_1, x_2, x_3) d\Omega \approx$$
$$\sum_{n_{int}^1=1}^{N_{int}^1} \sum_{n_{int}^2=1}^{N_{int}^2} \sum_{n_{int}^3=1}^{N_{int}^3} p(x_1(\xi\Big|_{n_{int}^1}), x_2(\xi\Big|_{n_{int}^2}), x_3(\xi\Big|_{n_{int}^3}))J(\xi\Big|_{n_{int}^3})W_{n_{int}^1} W_{n_{int}^2} W_{n_{int}^3} \tag{30}$$

Shape functions were used to interpolate quantities between the nodes of Ω_h but were defined in the parametric space $\square$. The functions used meet the constraint shown shown in Equation 31.

$$N_i(\xi_j) = \delta_{ij} \tag{31}$$

In Equation 31, δ_{ij} is the Kronecker delta, N_i is the shape function of the ith node, and ξ_j is the parametric coordinates of the jth node. Lagrange basis functions were used for shape functions for first order approximations; serendipity shape functions were used for second order approximations. Lagrange shape functions were formed by the method shown in Equation 32.

$$N_i(\xi) = \prod_{\substack{j=1 \\ j \neq i}}^{j=n_{en}} \frac{\xi - \xi_j}{\xi_i - \xi_j} \tag{32}$$

The number of nodes for each element is represented by n_{en} in Equation 32.

2.3 Linear Algebra

The entirety of the stiffness matrix was not stored as it is sparsely populated. Instead, the matrix was stored as a collection of arrays using compressed row storage. Each array represented a row in the stiffness matrix. The elements of each array stored the non-zero value and column index of the matrix element it represented. This project made use of the implementation available through `TPetra`.

The Generalized Minimal Residual (GMRES) method was used to solve the system described in Equation 24. Specifically the implementation from the `Belos` package was used. The GMRES is an iterative method; for this project a maximum iteration limit of 200 was used with a tolerance of 10^{-10}.

3 Code Description

The code written for this project was made in `C++` and makes use of the `APF`, `GMI`, and `Trilinos` libraries. The user interacts with the code at the command line and provides the following arguments (not including the executable name and path):

1. `<model_file>.dmg` which is the geometric model.

2. `<mesh_file>.smb` which is the mesh of the geometric model.

3. `<associations>.txt` which dictates and names sets of mesh nodes and entities to construct by classification on the geometric model. This later allows the user to declare boundary conditions by name in the `.yaml` file. An example `<associations>.txt` file is shown in Appendix B.1.

4. `<Problem_Statement>.yaml` which dictates material properties (Young's modulus and Poisson's ratio), boundary conditions (Dirichlet and Neumann), and parameters for the linear solver such as method and tolerance. An example `.yaml` file is shown in Appendix B.2.

5. `Order` as an integer value. It is the order of both the shape functions used and numerical integration. Only one and two are supported in this project.

6. `Body_Load` as three floating point numbers. It represent the force per unit area acting on the model as (X, Y, Z) vector. For the plane stress application, the Z component is not used.

The classes created for this code are presented in Section 3.1. This includes a brief description of the class's purpose, its public member variables, and public member functions. Extra utility features of classes are not shown in Section 3.1 for brevity. Following this in Section 3.2, are Algorithm Blocks which list the pseudo code for the entire project.

3.1 Class Description

The class shown in Listing 1 is the discretization class. It creates and stores sets of mesh entities and nodes that the user can later use when applying boundary conditions. It also creates the maps from the degrees of freedom of the nodes in the mesh to the rows in the linear algebra system. The header and source file for this class are shown in Appendix A.8 and A.9, respectively.

Listing 1: Discretization class member variables and functions.

```
class Disc{
  // Public member functions and descriptions:

  // Constructor:
  //  Creates the discretization object based on the mesh
  //  and associations file. Creates the sets of nodes
  //  and maps as well.
  Disc( mesh, assoc_file);

  // Gets the set of mesh sides by name.
  // Returns a standard vector of the
  // mesh entities.
  std::vector<apf::MeshEntity*>
    get_sides( std::string set_name);

  // Gets the set of mesh nodes by name.
  // Returns a standard vector of the
  // mesh nodes.
  std::vector<apf::Node*>
    get_node( std::string set_name);

  // Private member variables:

  // The mesh used for this problem.
  apf::Mesh* mesh;

  // The number of spatial dimensions.
  int num_dims;

  // A map from node set name to the set of nodes.
  std::map<std::string, std::vector<apf::Node> > node_sets;
```

```
  // A map from side set name to the set of
  // mesh entities.
  std::map<std::string, std::vector<apf::MeshEntity*> > side_sets;
};
```

The class shown in Listing 2 is the Finite Element Solver class. It creates, stores, and solves the linear system which represents the finite element problem. Creating the system includes creating the global stiffness matrix and assigning boundary conditions to create the correct stiffness matrix and forcing vector for the given problem statement and mesh. It then creates assigns the solution and secondary variables to a field. This field can then be written to Vtk files which are then viewed in ParaView. It can also calculate the Root Mean Square (RMS) error of the approximated displacement solution. The header and source file for this class are shown in Appendix A.12 and A.13, respectively.

Listing 2: FESolver class members variables and functions.

```
class FESolver{
  // Public member functions and descriptions:
  public:

  // Constructor:
  //  Constructs the solver object based on the maps and
  //  sets created by the discretization object. Also
  //  stores data from the .yaml (parameterList),
  //  solution order, and body load.
  FESolver( disc, parameterList, order, load);

  // Fill in the global stiffness matrix, adjusts and
  // creates the forcing vector by applying boundary
  // conditions and body loads. The details of this
  // function are shown in Algorithm 5
  void solve();

  // Set the solution vector, displacement, to an
  // apf::field. The details of this function are
  // shown in Algorithm 12.
  void set_disp_to_field( field);

```

```
  // Set the force vector to an apf::field.
  // The details of this function are shown in
  // Algorithm 12.
  void set_force_to_field( field );

  // Calculates and sets the stress to an apf::field.
  // Details of this function are shown in
  // Algorithm 13.
  void set_stress_to_field( field );

  // Calculate the Root Mean Square error in the
  // solution by comparing to the analytical solution.
  // Details of this function are shown in
  // Algorithm 14.
  double get_error();

  // Private member functions and descriptions:
  private:

  // Assembles the global stiffness matrix.
  // Does not consider boundary conditions.
  // The details of this function are shown in
  // Algorithm 6.
  void assemble_LHS();

  // Assemble the forcing vector.
  // Considers Dirichlet and Neumann boundary
  // conditions and body loads. Also makes
  // needed modifications to stiffness matrix
  // for Dirichlet conditions.
  // The details of this function are shown in
  // Algorithm 7.
  void assemble_RHS();

  // Private variables and descriptions:

  // Pointer to the discretization object
  Disc* disc;

  // Pointer to the parameter list that contains
```

```
  // material properties and boundary conditions.
  ParameterList* params;

  // The linear algebra object. This class is described
  // in Listing 3.
  LinAlg* la;

  // The order of accuracy. Used for order of shape
  // functions and numerical integration.
  int order;

  // The array of body load components.
  double g[3];

  // Pointer to the elemental stiffness integrator.
  ElasticStiffness* LHS;
};
```

The class shown in Listing 3 is the Linear Algebra class. This class is an interface to the global stiffness matrix, forcing vector, and solution vector. The stiffness matrix uses the compressed row storage matrix class from the `Tpetra` software package. The forcing vector and solution vector are stored as defined by the Vector class also from the `Tpetra` software package. The header and source file for this class are shown in Appendix A.14 and A.15, respectively.

Listing 3: Linear Algebra class member variables and functions.

```
class LinAlg{
  public:
  // Public member functions and descriptions:

  // Constructor:
  //  Allocates space for the stiffness matrix, K,
  //  solution vector, U, and force vector, F, based
  //  on the maps created from the discretization
  //  object.
  LinAlg( disc);

  // Public member variables and descriptions:
  // Type definitions used for brevity.

```

```
  // The stiffness matrix.
  Matrix K;

  // The solution vector.
  Vector U;

  // The forcing vector.
  Vector F;
};
```

The class shown in Listing 4 is the Elemental stiffness integrator class. This class inherits from the `apf::Integrator` class which provided the frame work for processing mesh elements. This member functions of this class dictate the operations that need to be done for each element and each integration point of each element to create the elemental stiffness matrix. The header and source file for this class are shown in Appendix A.10 and A.11, respectively.

Listing 4: Integrator class for elemental stiffness matrices.

```
class ElasticStiffness{
  public:
  // Public member functions and descriptions.

  // Constructor:
  //  Creates the integrator object based on the mesh,
  //  order of integration, and material properties.
  ElasticStiffness( mesh, order, E, nu);

  // Prepares each new mesh element for evaluation.
  void inElement( apf::MeshElement* element);

  // Updates the elemental stiffness matrix for each
  // integration point. Takes the parametric location,
  // weight, and differential volume of
  // the integration point. The details of this
  // function are shown in Algorithm 10.
  void atPoint( apf::Vector3 para, double w, double dv);

  // Finalizes and frees data once done with each
  // element.
  void outElement();
```

```

  // Public member variables and descriptions:

  // The elemental stiffness matrix.
  apf::DynamicMatrix Ke;

  private:
  // Private member variables and descriptions:

  // Number of spatial dimensions
  int num_dims;

  // Number of nodes per element
  int num_elem_nodes;

  // Number of degrees of freedom per element
  int num_elem_dofs

  // Material stiffness matrix. Plane stress is used.
  apf::DynamicMatrix D;

  // Gradient of shape functions, product with D, and transpose.
  apf::DynamicMatrix B;
  apf::DynamicMatrix DB;
  apf::DynamicMatrix BT;

  // Temporary stiffness matrix.
  apf::DynamicMatrix K_tmp;

  // The mesh, basis function shape, and current element.
  apf::Mesh* mesh;
  apf::FieldShape shape;
  apf::MeshElement* mesh_element;
};
```

The class shown in Listing 5 is the Elemental traction integrator class. This class inherits from the `apf::Integrator` class which provided the frame work for processing mesh elements. This member functions of this class dictate the operations that need to be done for each element and each integration point of each element to create the elemental force vector contributions. This class works for only one element and one component of the traction at at time. This means that if a traction with components of $(t_x, t_y, 0)$ was prescribed on a boundary, this class would need to be used for each element on the boundary twice; once for t_x and once for t_y. The header and source file for this class are shown in Appendix A.18 and A.19, respectively.

Listing 5: Integrator class for elemental force vectors from tractions.

```
class elemTrac{
  public:
  // Public member functions and descriptions.

  // Constructor:
  //  Creates the integrator object based on the mesh,
  //  order of integration, traction value, and the
  //  corresponding spatial dimension.
  elemTrac( mesh, order, value, eqNum);

  // Prepares each new mesh element for evaluation.
  void inElement( apf::MeshElement* element);

  // Updates the elemental force vector for each
  // integration point. Takes the parametric location,
  // weight, and differential volume of
  // the integration point. The details of this
  // function are shown in Algorithm 9.
  void atPoint( apf::Vector3 para, double w, double dv);

  // Finalizes and frees data once done with each
  // element.
  void outElement();

  // Public member variables and descriptions:

  // The elemental force vector.
  apf::DynamicVector fe;

  private:
  // Private member variables and descriptions:

  // Number of spatial dimensions
```

```
  int num_dims;

  // Number of nodes per element
  int num_elem_nodes;

  // Number of degrees of freedom per element
  int num_elem_dofs

  // Value of this traction component
  double value;

  // The mesh, basis function shape, and current element.
  apf::Mesh* mesh;
  apf::FieldShape shape;
  apf::MeshElement* mesh_element;
};
```

The class shown in Listing 6 is the Elemental body load integrator class. This class inherits from the `apf::Integrator` class which provided the frame work for processing mesh elements. This member functions of this class dictate the operations that need to be done for each element and each integration point of each element to create the elemental force vector contributions due to a body load. The header and source file for this class are shown in Appendix A.2 and A.3, respectively.

Listing 6: Integrator class for elemental force vectors from body loads.

```
class BodyLoad{
  public:
  // Public member functions and descriptions.
  // Constructor:
  //  Creates the integrator object based on the mesh,
  //  order of integration, and body load vector.
  elemTrac( mesh, order, g);

  // Prepares each new mesh element for evaluation.
  void inElement( apf::MeshElement* element);

  // Updates the elemental force vector for each
  // integration point. Takes the parametric location,
  // weight, and differential volume of
```

```
  // the integration point. The details of this
  // function are shown in Algorithm 8.
  void atPoint( apf::Vector3 para, double w, double dv);

  // Finalizes and frees data once done with each
  // element.
  void outElement();

  // Public member variables and descriptions:

  // The elemental force vector contribution
  apf::DynamicVector fe;

  private:
  // Private member variables and descriptions:

  // Array of shape function values.
  apf::NewArray<double> N;

  // The order of numerical integration accuracy
  int order;

  // Number of spatial dimensions
  int num_dims;

  // Number of nodes per element
  int num_elem_nodes;

  // Number of degrees of freedom per element
  int num_elem_dofs

  // Vector of the body load.
  apf::Vector3 g;

  // The mesh, basis function shape, and current element.
  apf::Mesh* mesh;
  apf::FieldShape shape;
  apf::MeshElement* mesh_element;
};
```

3.2 Pseudo Code

The main code is shown in Algorithm 1. For all algorithms presented in the this section, namespaces are not shown for brevity. Algorithm blocks which show the pseudo codes for subroutines written for this project are shown after Algorithm 1. Pseudo code for functions and methods not written for this project but borrowed from other libraries, like `PUMI`, `APF`, and `Trilinos`, are not shown.

Algorithm 1: The main function for this project. The function takes 10 arguments which are described in section 1.

```
main (int argc, char ** argv)
  /* Create a parameter list object and populate with
     information from yaml file.                                  */
  ParameterList p;
  updateParametersFromYaml( <yamlFile>, p);
  /* Load the mesh from file.                                     */
  m = loadMdsMesh( <modelFileName>.dmg, <meshFileName>.smb);
  /* Create a discretization-class object based on the mesh and
     the associations file.                                       */
  d = createDiscretization( m, <associationsFile>);
  /* Create a solver-class object, then solve the FE system.
     The system is solved using the GMRES method discussed in
     subsection 2.3 using the Belos software package.             */
  s = createSystemSolver( d, p, integration_order, body_load);
  s→ solve();
  /* Create and populate fields with solution information.
     Write to Vtk file. (Note: 3 is the dimension in
     writeVtkFiles.)                                              */
  s→ set_displacement_to_field();
  s→ set_traction_to_field();
  s→ set_stress_to_field();
  writeVtkFiles( <FileName>, m, 3);
  /* Calculate the RMS of the error and report to user.           */
  s→ get_error();
  /* Allocated memory space is freed.                             */
  return 0 ;
```

Algorithm 2: Constructor for the discretization object.

```
createDiscretization (mesh * m, char* <associationsFile>)
/* Verify the mesh is usable; abort program otherwise.      */
m→ verify();
/* Compute maps and node sets for creating the linear algebra
   data types and assigning boundary conditions.            */
compute_maps();
compute_sets();
return;
```

Algorithm 3: Member function of the discretization object that constructs the maps needed to create the sparse stiffness matrix, solution, and forcing vector.

```
discretization::compute_maps ()
get_number_nodes( mesh);
numbering = create_new_numbering( mesh);
for Each Node do
    /* Assign each node a unique global identification number.
                                                              */
    assign_node_ID( this_node, numbering);
end
create_map_outline( numbering);
for Each Node do
    for Each Nodal Degree of Freedom do
        assign_map_value();
    end
end
return;
```

Algorithm 4: Member function of the discretization object that constructs node sets for assigning boundary conditions.

```
discretization::compute_sets ()
for Each Set Declared in yaml do
    for Each Mesh Entity of Defined Dimension do
        get_classification( mesh_entity);
        if Geometric Entity Needs Set From yaml then
            /* Add this mesh entity to a list for this set.    */
        end
    end
end
return;
```

Algorithm 5: Method for creating and then solving the finite element system.

```
solver::solve ()
/* Assemble the unreduced stiffness matrix and forcing vector.
                                                              */
assemble_Left_Hand_Side();
assemble_Right_Hand_Side();
/* Apply Dirichlet boundary conditions to the system.        */
apply_Dirichlet_BCs();
/* Solve the Kd = F system iteratively using GMRES method.
   */
solve_linear_system();
return;
```

Algorithm 6: Method for creating the unreduced stiffness matrix accounting for the final system begin sparsely populated.

```
solver::assemble_Left_Hand_Side ()
get_material_properties();
for Each Mesh Element do
    integrate_elastic_stiffness();
    get_element_node_IDs();
    for Each Nodal Degree of Freedom do
        get_global_row_number();
        sum_elemental_into_global_stiffness();
    end
end
return;
```

Algorithm 7: Method for creating the forcing vector based on Neumann boundary conditions and body loads.

```
solver::assemble_Right_Hand_Side ()
/* Neumann boundary conditions are first evaluated, then body
   loads.  This order is arbitrary.                           */
for Each Neumann Boundary Condition from yaml do
    get_traction_vector();
    /* The sets computed earlier in Algorithm 4 are used here
       to quickly retrieve the list of mesh entities on the
       boundary that need to be evaluated.                    */
    get_boundary_mesh_entities();
    for Each Mesh Entity do
        integrate_traction();
        get_element_node_IDs();
        for Each Nodal Degree of Freedom do
            get_global_row_number();
            sum_elemental_into_global_forcing();
        end
    end
end
for Each Mesh Region do
    integrate_body_load();
    get_element_node_IDs();
    for Each Nodal Degree of Freedom do
        get_global_row_number();
        sum_elemental_into_global_forcing();
    end
end
return;
```

Algorithm 8: Integration method to construct elemental forcing vector contributions due to Neumann boundary conditions.

```
integrate_body_load ()
for Each Integration Point do
    p = get_parametric_location();
    w = get_point_weight(p);
    dv= get_det_J(p);
    N = get_basis_func(p);
    for Each Element Node do
        for Each Degree of Freedom do
            f_tmp ← f_tmp + N ∗ g ∗ w ∗ dv ;
            /* Where "g" is the vector of body loads.          */
        end
    end
    F_e ← F_e + f_tmp;
end
return;
```

Algorithm 9: Integration method to construct elemental forcing vector contributions due to Neumann boundary conditions.

```
integrate_traction ()
for Each Integration Point do
    p = get_parametric_location();
    w = get_point_weight(p);
    dv= get_det_J(p);
    N = get_basis_func(p);
    for Each Element Node do
        for Each Degree of Freedom do
            f_tmp ← f_tmp + N ∗ T ∗ w ∗ dv ;
            /* Where "T" is the vector of tractions.           */
        end
    end
    F_e ← F_e + f_tmp;
end
return;
```

Algorithm 10: Integration method to construct elemental stiffness matrices.

integrate_elastic_stiffness ()

D = fill_material_elasticity_tensor();

for *Each Integration Point* **do**

  p = get_parametric_location();

  w = get_point_weight(p);

  dv= get_det_J(p);

  B = get_basis_func_gradient(p);

  $k_{tmp} \leftarrow B^T * D * B * w * dv$;

  $K_e \leftarrow K_e + k_{tmp}$;

end

return;

Algorithm 11: Method for adjusting stiffness matrix and force vector for Dirichlet boundary conditions.

```
solver::apply_Dirichlet_BCs ()
for Each Dirichlet Boundary Condition from yaml do
    get_displacement_vector();
    /* The sets computed earlier in Algorithm 4 are used here
       to retrieve the list of mesh entities on the boundary
       that need to be evaluated.                                */
    get_boundary_mesh_entities();
    for Each Mesh Entity do
        get_element_node_IDs();
        for Each Nodal Degree of Freedom do
            /* All row entries but the diagonal term for this row
               of the stiffness matrix are set to zero.  The
               diagonal term is set to one and the old diagonal
               term is multiplied by the displacement component
               for this degree of freedom and placed into the
               forcing vector.                                   */
            tmp = get_global_stiffness_diagonal();
            for Each Non-Zero Row Entry do
                if row == column then
                    /* Set stiffness diagonal to one.  Set force to
                       the product of the displacement and
                       stiffness.                                */
                    set_stiffness(row, column, 1.0);
                    set_force( row, tmp*displacement);
                end
                else
                    /* Set off-diagonal components of row to zero.
                                                                 */
                    set_stiffness(row, column, 0.0);
                end
            end
        end
    end
end
return;
```

Algorithm 12: Generic function for setting a vector solution to a field that can then be written to a Vtk file for use with ParaView.

```
solver::set_< * >_to_field ()
get_mesh_nodes(); for Each Node do
    for Each Nodal Degree of Freedom do
        /* The value that the field represents (displacement,
           traction, or stress) is gotten here.                */
        get_solution_component();
        set_component_to_field();
    end
end
return;
```

Algorithm 13: Calculates the Cauchy stress and assigns matrix to field that can then be written to a Vtk file for use with ParaView.

```
set_stress_to_field ()
for Each Mesh Element do
    /* Get the displacement vector for this element.          */
    u_e = get_elemental_solution();
    for Each Integration Point do
        p = get_parametric_location();
        B = get_basis_func_gradient(p);
        sigma ← E * B * u_e ;
    end
    update_field( sigma);
end
return;
```

Algorithm 14: Calculates the Root Mean Square error of the solution vector as compared to the analytical solution.

```
get_error ()
e = new_vector(); for Each Mesh Node do
    for Each Spatial Dimension do
        row = get_DOF_ID();
        /* The analytical solution is discussed further in
           Section 4.                                        */
        a = get_analytical_solution();
        e[row] = a - u_e[row];
    end
end
RMS ← (e→norm_2())/sqrt(e→length());
```

4 Tests & Results

The code written for this project was tested with triangular and quadrilateral elements. Each type of element was also tested in a linear and quadratic level of accuracy; linear in that shape functions and numerical integration order were linear - quadratic in that the shape functions and numerical integration order were quadratic. The same 1 by 1 unit length model was used for all tests for consistency. The same material properties were used all all tests as well. Young's modulus was set to 1000.0 and Poisson's ratio was set to 0.25. The same base Dirichlet boundary conditions were used for all tests as well; the leftmost and bottommost edges were constrained to perpendicular motion as shown in Figure 1. Each element type and accuracy level combination was tested with the following three test numbers:

1. An additional Dirichlet boundary condition on the rightmost edge of 0.01.

2. A Neumann boundary condition on the rightmost edge of 100.0.

3. A body load of $(10.0, 0.0, 0.0)$ where the first component is positive to the left in Figure 1.

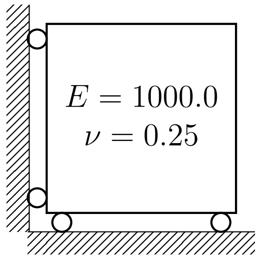

Figure 1: Base Dirichlet boundary conditions with material properties shown.

Images from these three tests are shown in Sections 4.1 and 4.2 for linear and quadratic triangular elements, respectively. This is then repeated for quadrilateral elements; results from linear quadrilateral tests are shown in Section 4.3 results from quadratic quadrilateral tests are shown in Section 4.4. Additionally, solution convergence is shown for the linear triangular elements in Section 4.1 for the body loading case. Results are presented graphically as prepared in ParaView.

4.1 Linear Triangular Elements

The displacement results for test 1 are shown in Figures 3 and 4. This displacement then caused the stress distribution shown in Figure 5. The original mesh used is shown in Figure 2. This mesh is also used for the quadratic triangular elements.

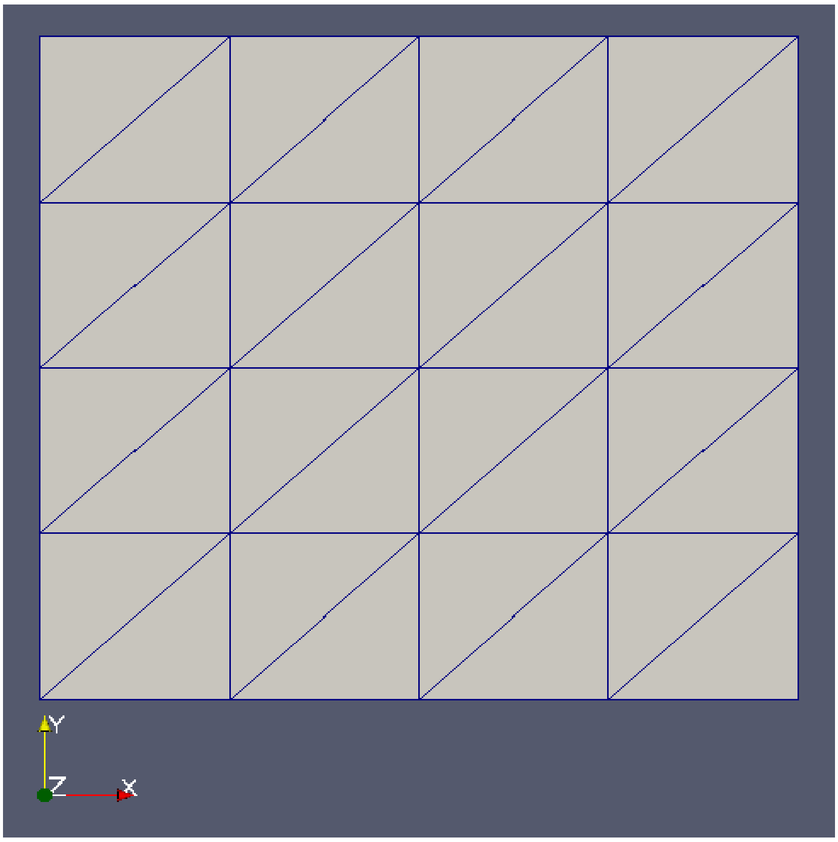

Figure 2: Original mesh used for test 1 and 2 for the linear and quadratic triangular elements.

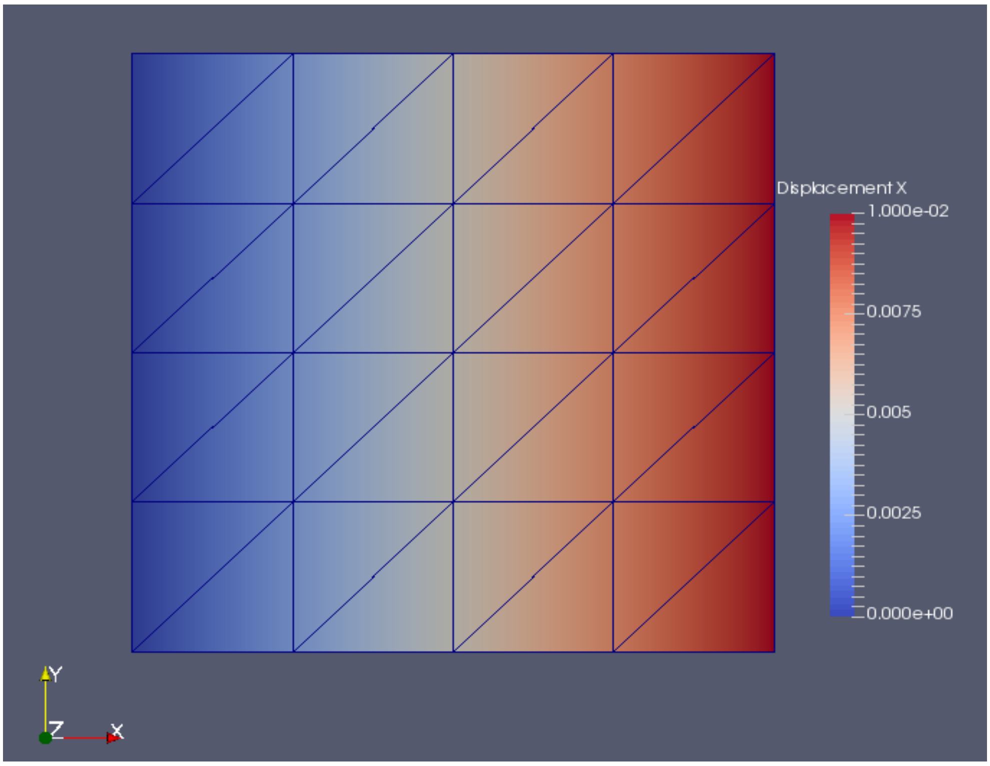

Figure 3: Deformation X component for test 1 of linear triangular elements.

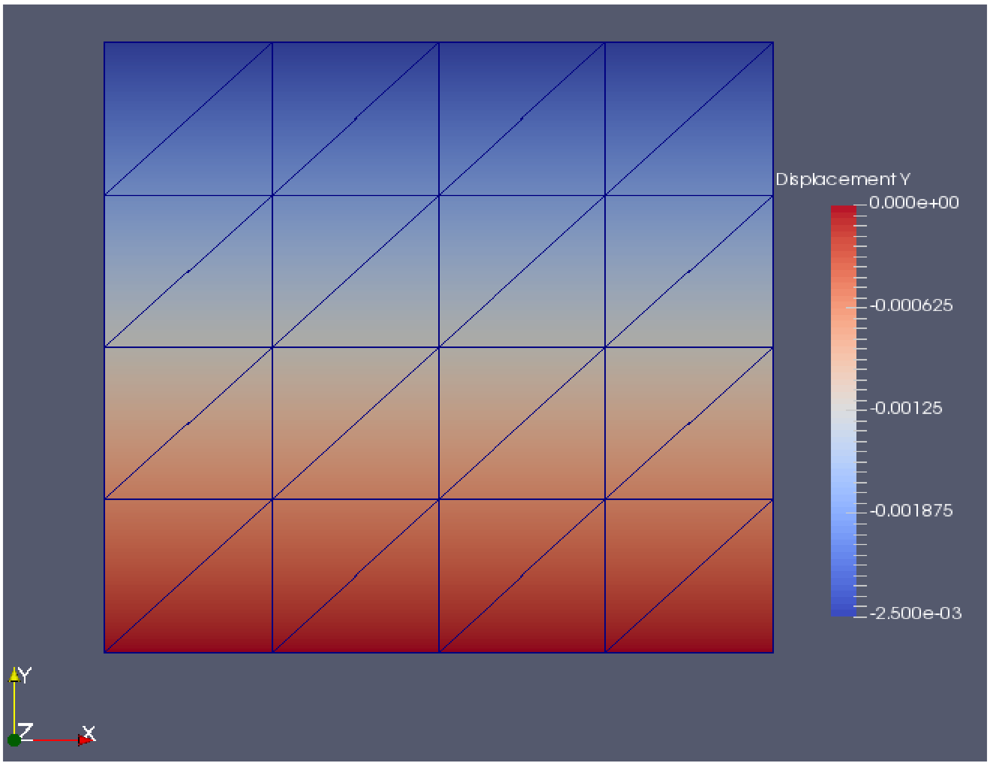

Figure 4: Deformation Y component for test 1 of linear triangular elements.

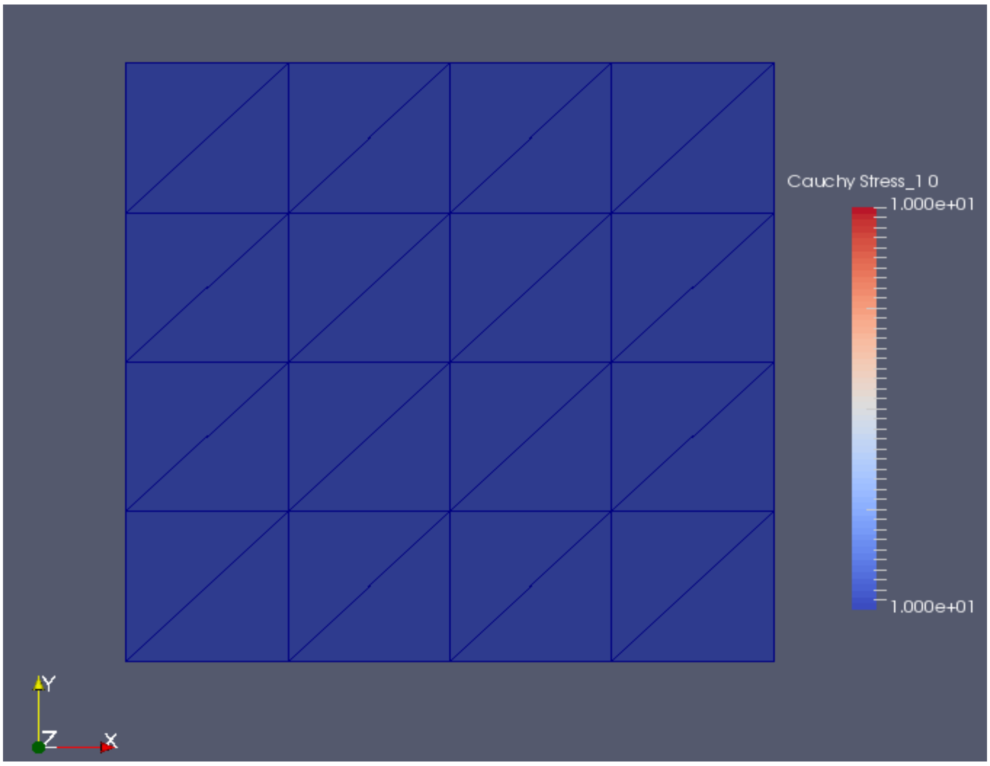

Figure 5: The xx component of the Cauchy stress for test 1. All other components were zero.

The displacement results for test 2 are shown in Figures 6 and 7. This displacement then caused the stress distribution shown in Figure 8.

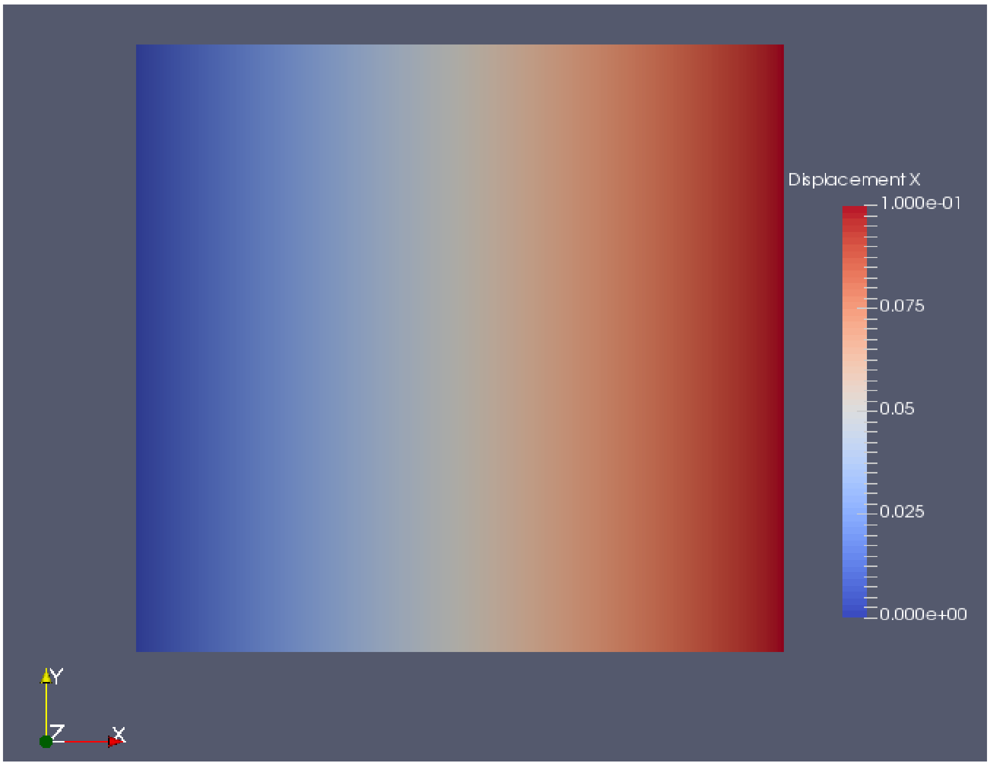

Figure 6: Deformation X component for test 2 of linear triangular elements.

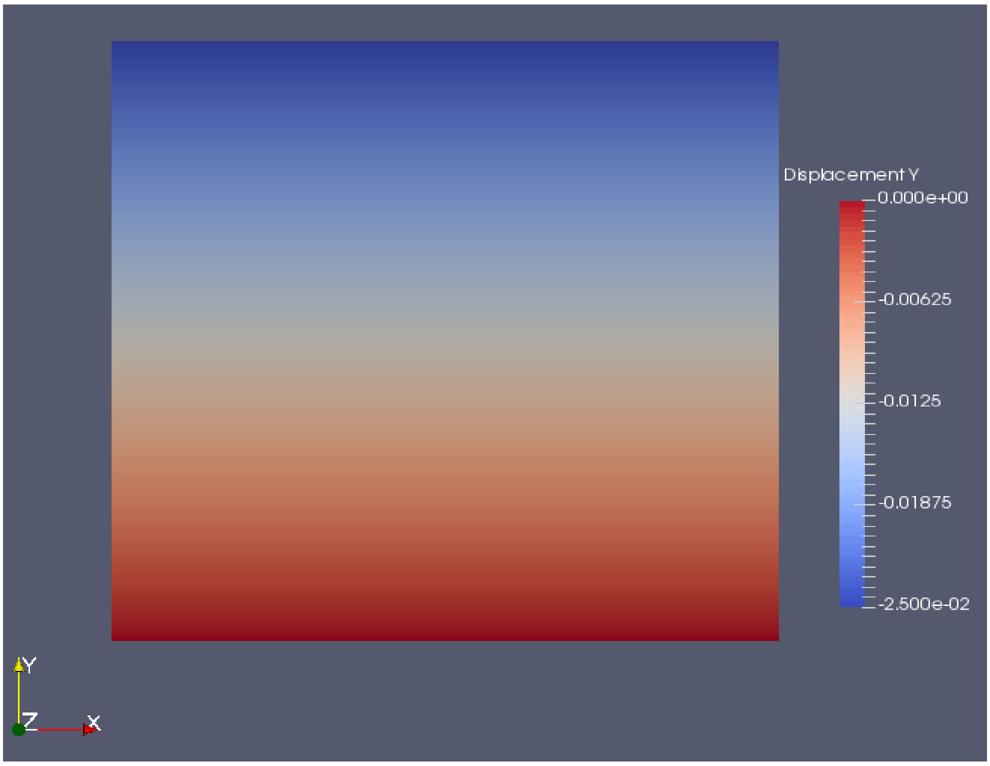

Figure 7: Deformation Y component for test 2 of linear triangular elements.

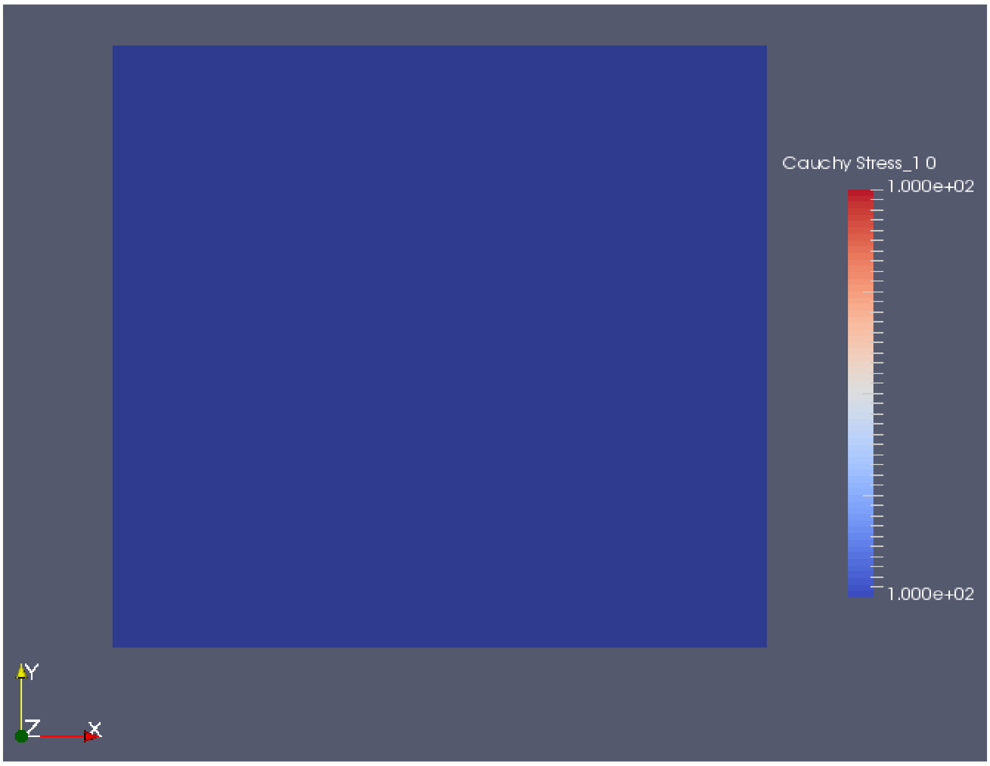

Figure 8: The xx component of the Cauchy stress for test 2. All other components were zero.

The displacement results for test 3 are shown in Figures 9 and 10. This displacement then caused the stress distribution shown in Figure 11.

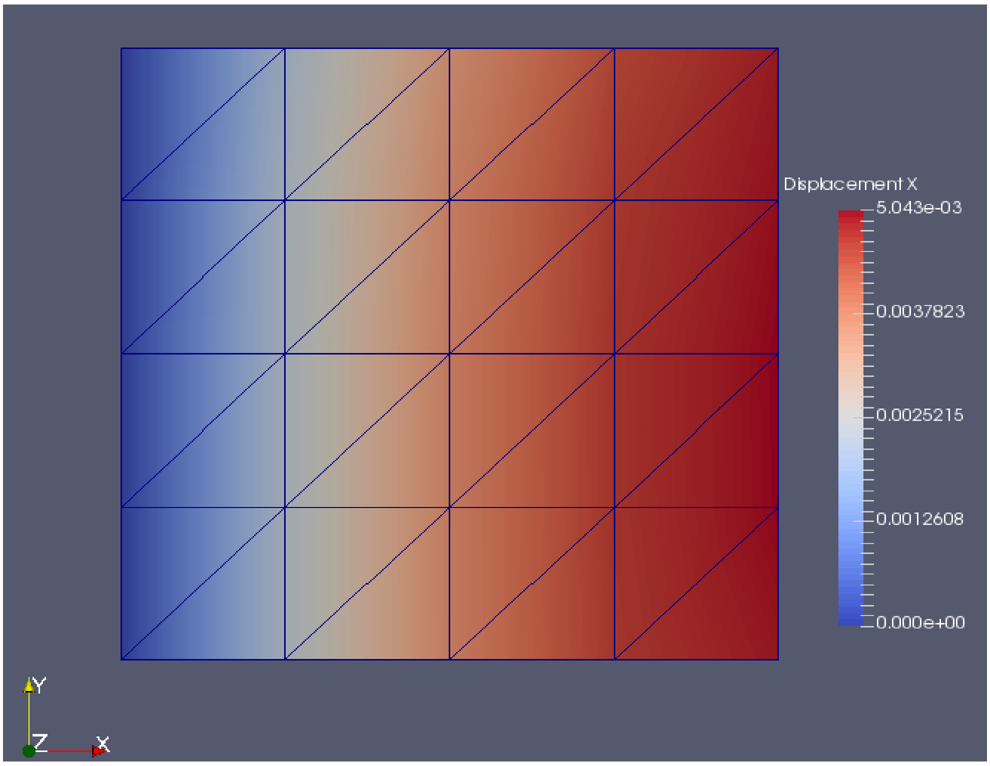

Figure 9: Deformation X component for test 3 of linear triangular elements.

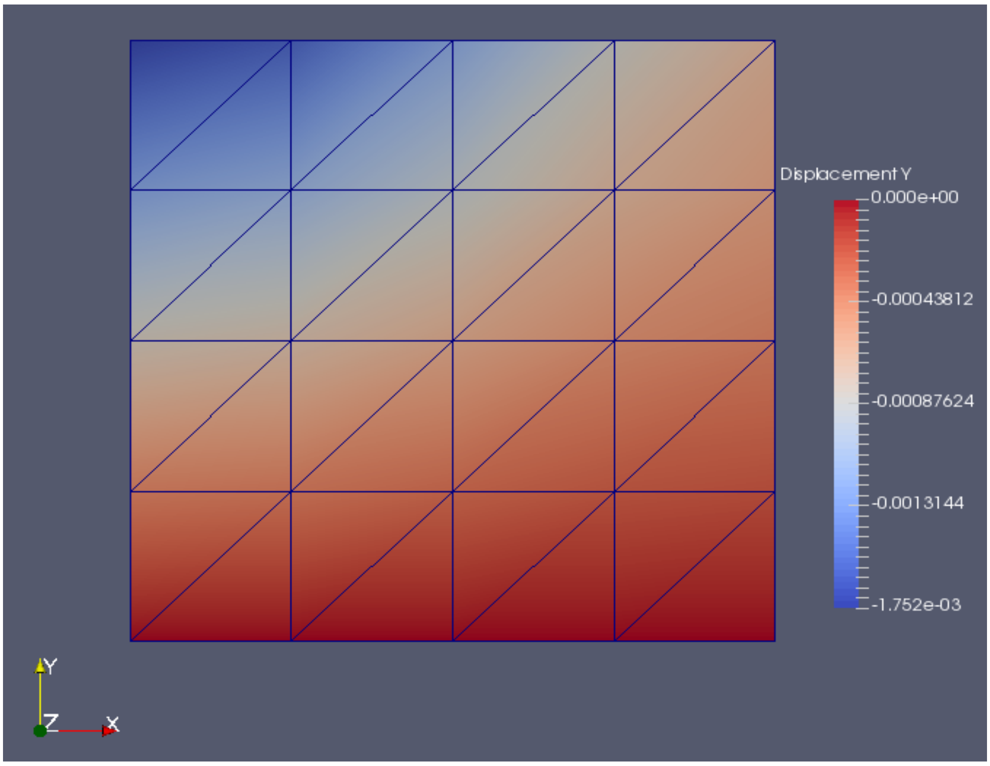

Figure 10: Deformation Y component for test 3 of linear triangular elements.

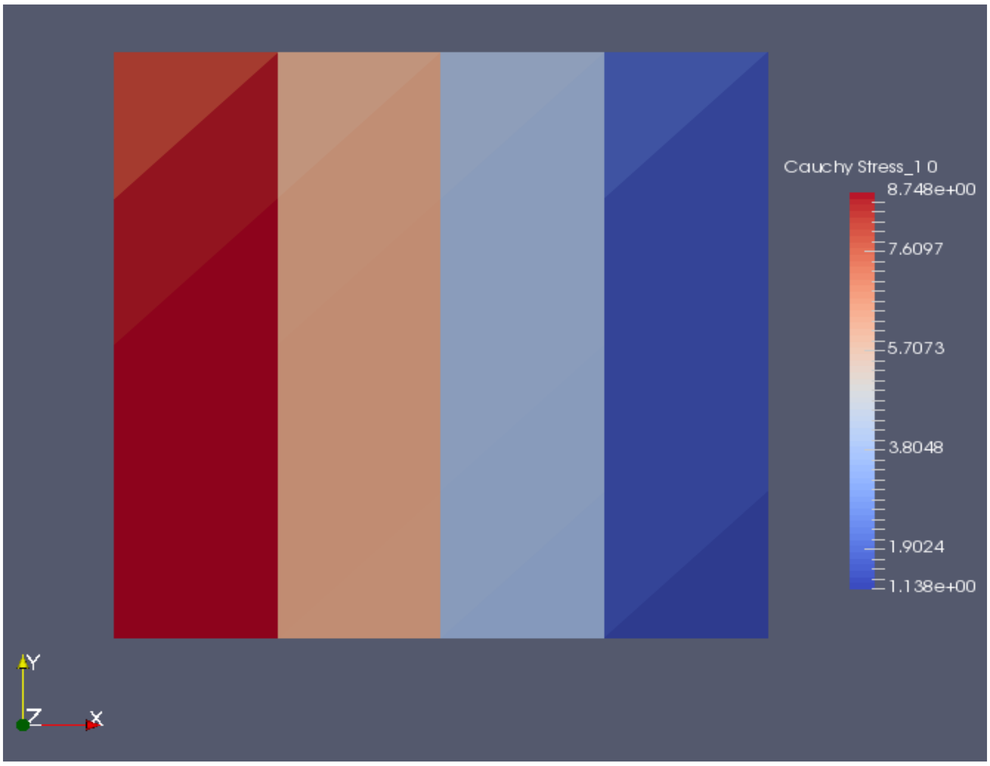

Figure 11: The xx component of the Cauchy stress for test 3.

Convergence was also measured using test 3. This is shown in Figure 12. The analytical solution for the body load case is shown in Equation 33:

$$U_a = \frac{g(a - x_1)x_1}{E} \begin{bmatrix} 1 \\ -\nu \\ 0 \end{bmatrix} \tag{33}$$

where U_a is the analytical displacement solution, a is the width of the plate (taken as 1 in this test), and x_1 is the position along the width of the plate. The origin is taken to be the bottom left corner.

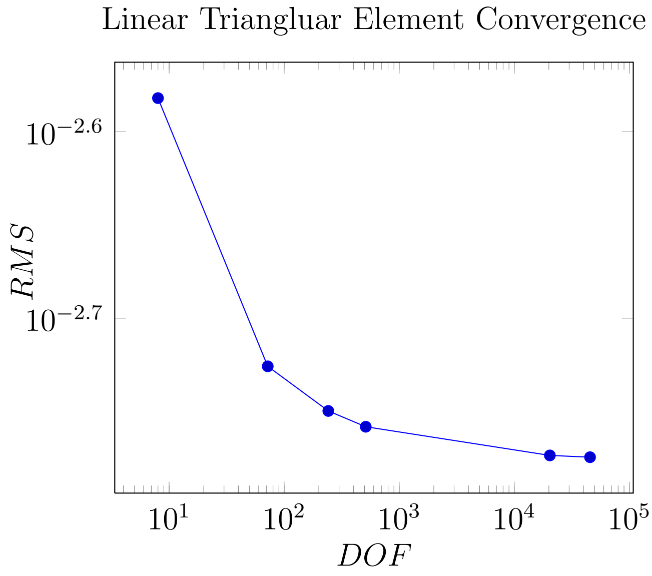

Figure 12: RMS of displacement error when compared to analytical solution with respect to increasing degrees of freedom.

As shown in Figure 12, the linear triangular elements are able to converge on the exact, quadratic, solution even though each element is only linear in accuracy.

4.2 Quadratic Triangular Elements

The displacement results for test 1 are shown in Figures 13 and 14. This displacement then caused the stress distribution shown in Figure 15. The original mesh used is shown in Figure 2.

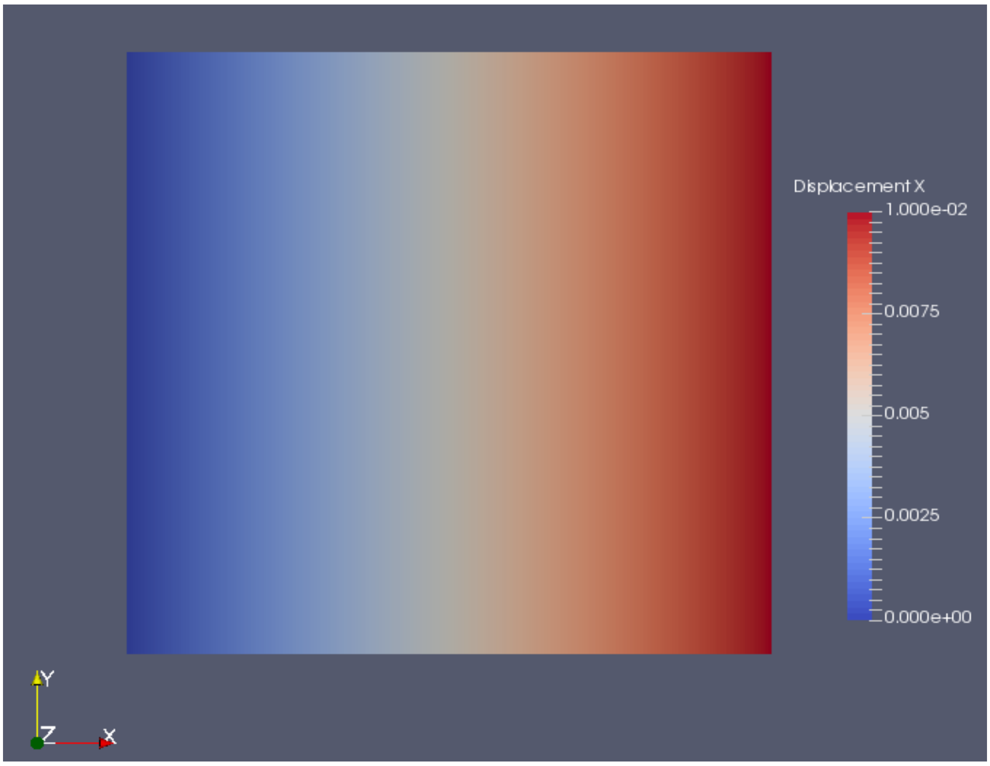

Figure 13: Deformation X component for test 1 of quadratic triangular elements.

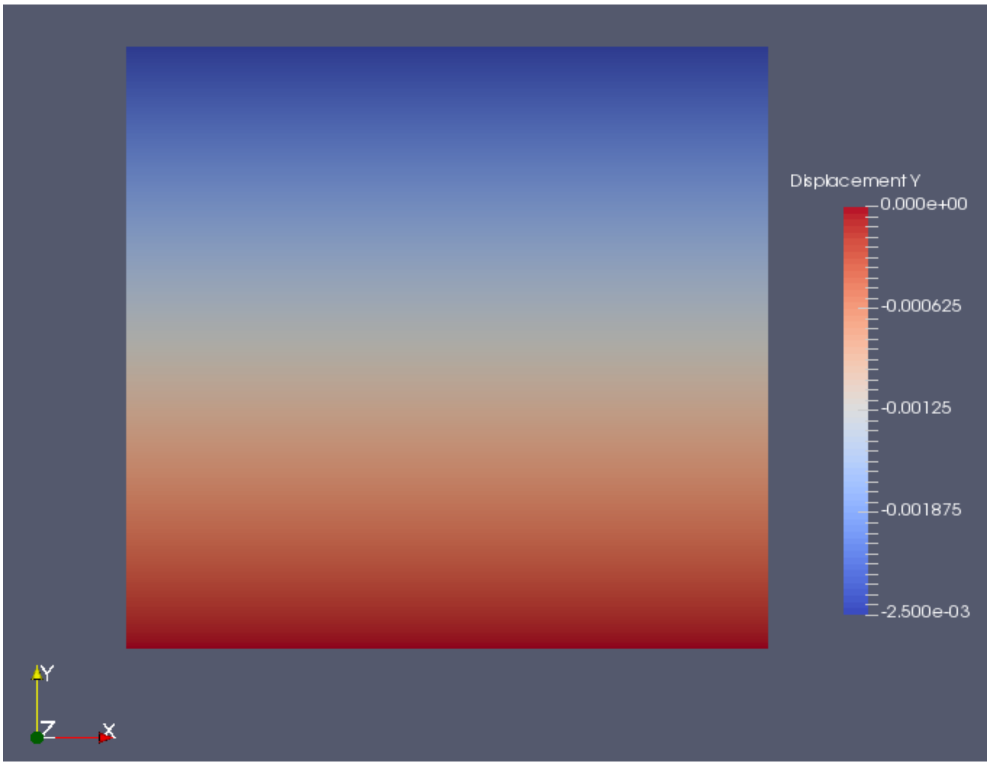

Figure 14: Deformation Y component for test 1 of quadratic triangular elements.

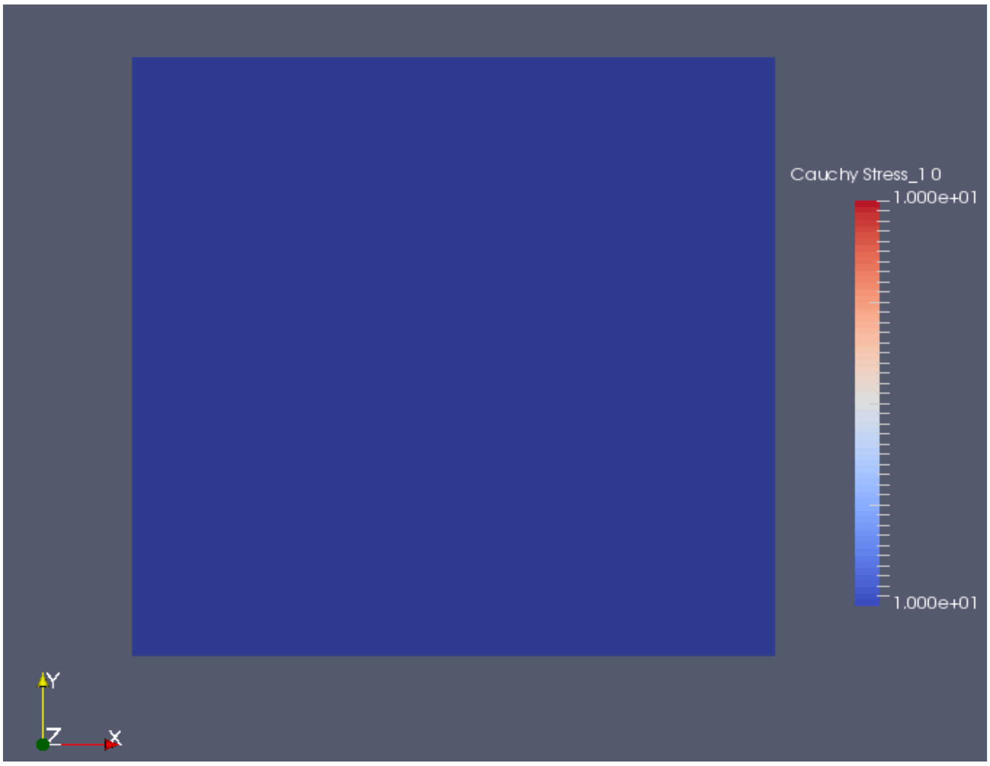

Figure 15: The xx component of the Cauchy stress for test 1. All other components were zero.

The displacement results for test 2 are shown in Figures 16 and 17. This displacement then caused the stress distribution shown in Figure 18.

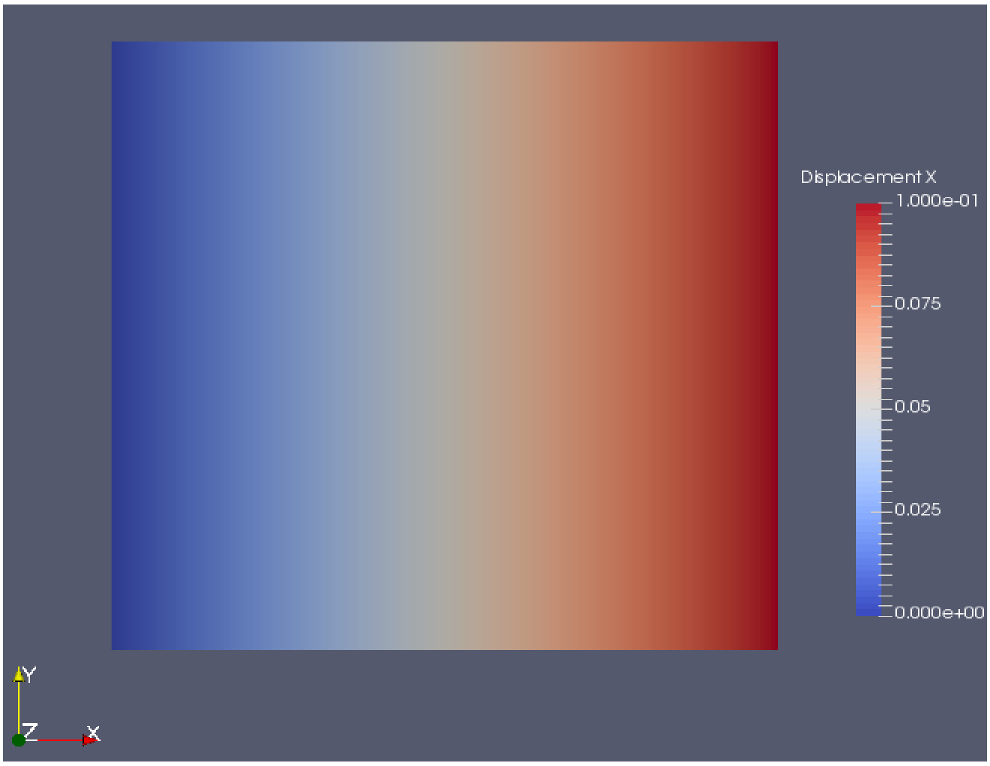

Figure 16: Deformation X component for test 2 of quadratic triangular elements.

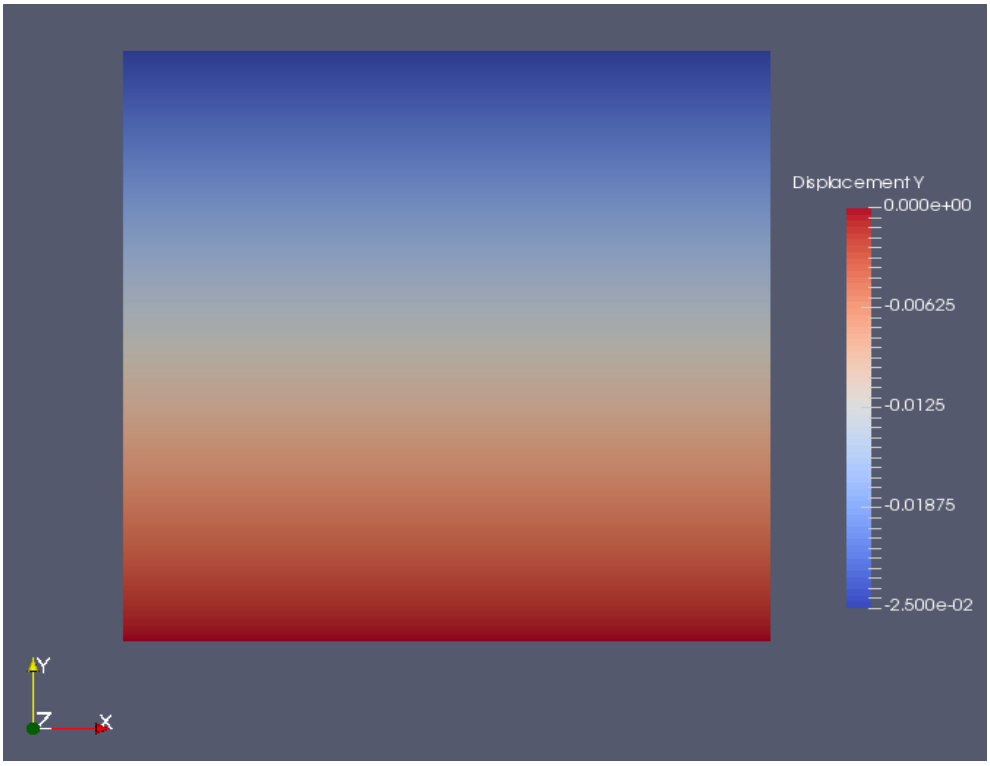

Figure 17: Deformation Y component for test 2 of quadratic triangular elements.

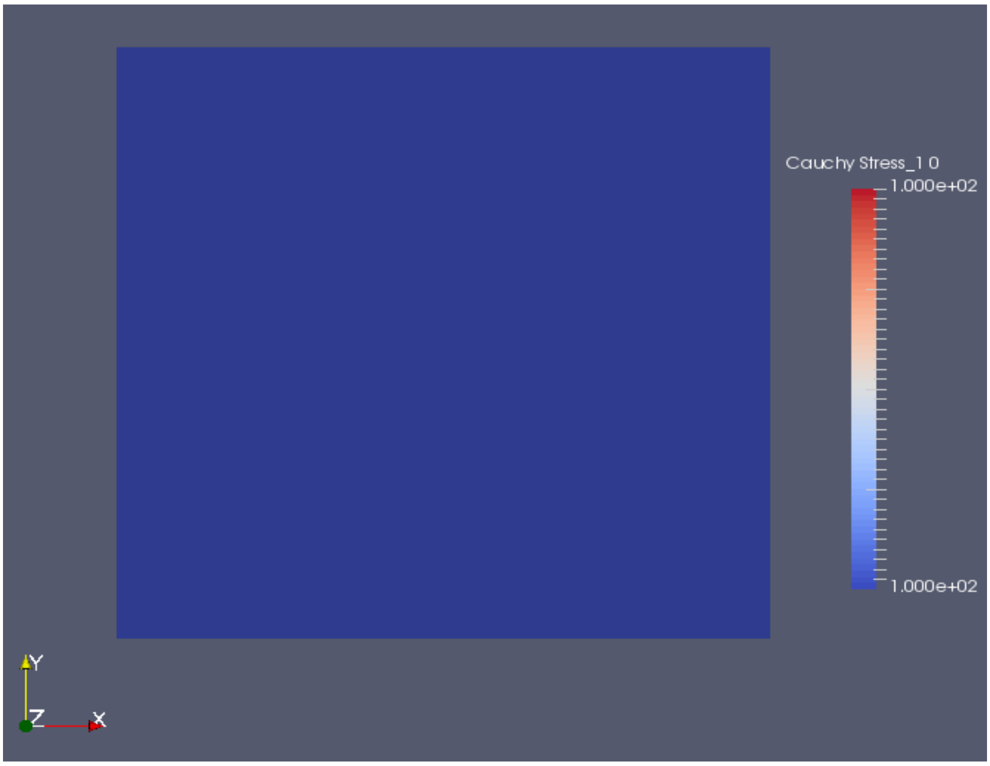

Figure 18: The xx component of the Cauchy stress for test 2. All other components were zero.

The displacement results for test 3 are shown in Figures 19 and 20. This displacement then caused the stress distribution shown in Figure 21.

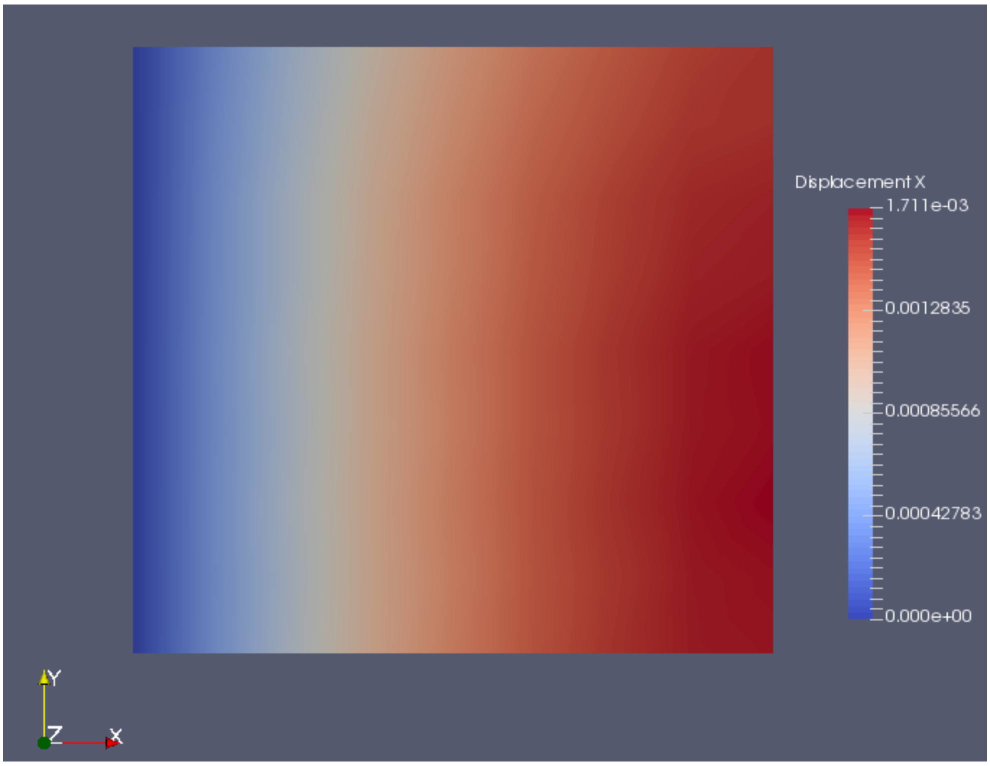

Figure 19: Deformation X component for test 3 of quadratic triangular elements.

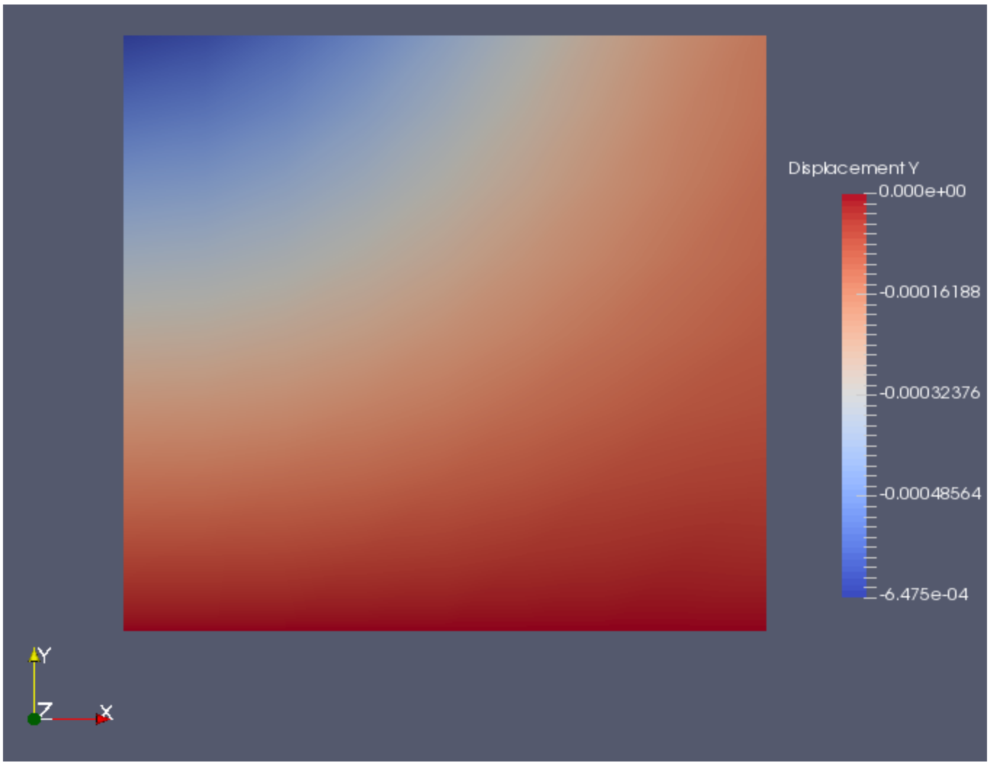

Figure 20: Deformation Y component for test 3 of quadratic triangular elements.

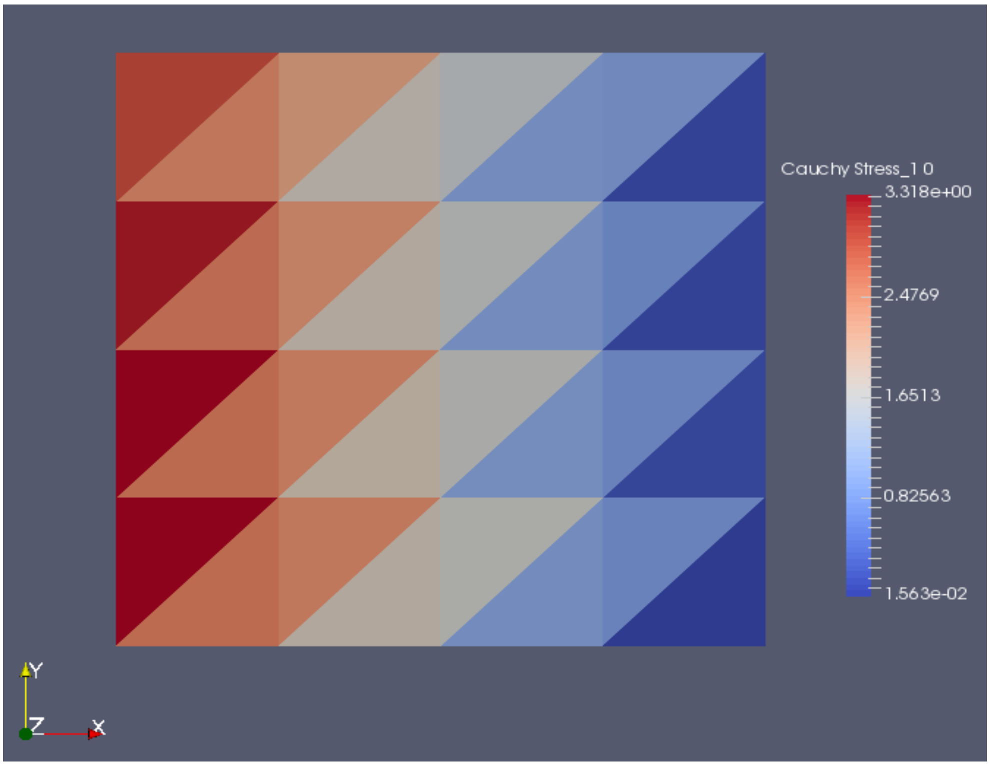

Figure 21: The xx component of the Cauchy stress for test 3.

4.3 Linear Quadrilateral Elements

The displacement results for test 1 are shown in Figures 23 and 24. This displacement then caused the stress distribution shown in Figure 25. The original mesh used is shown in Figure 22. This mesh is also used for the quadratic quadrilateral elements.

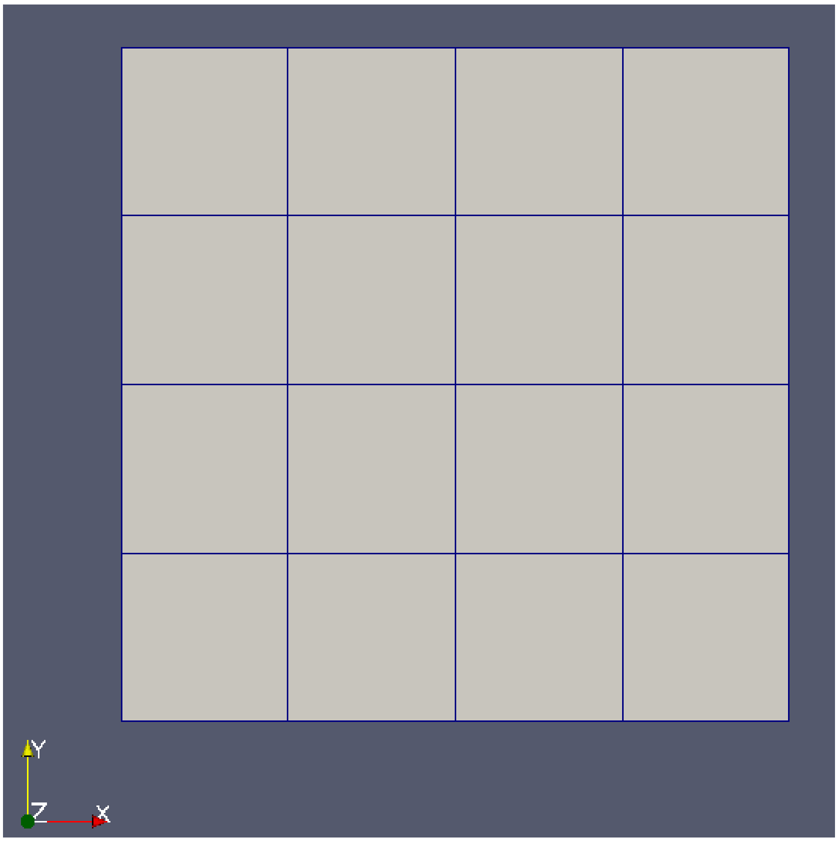

Figure 22: Original mesh used for test 1 and 2 for the linear and quadratic quadrilateral elements.

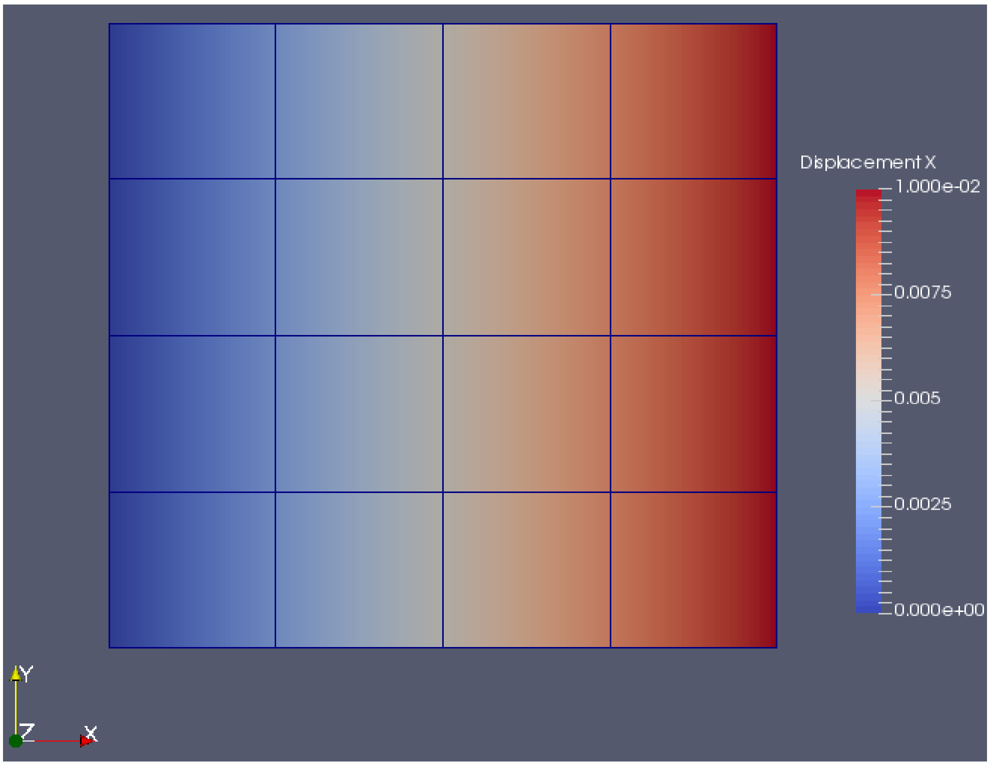

Figure 23: Deformation X component for test 1 of linear quadrilateral elements.

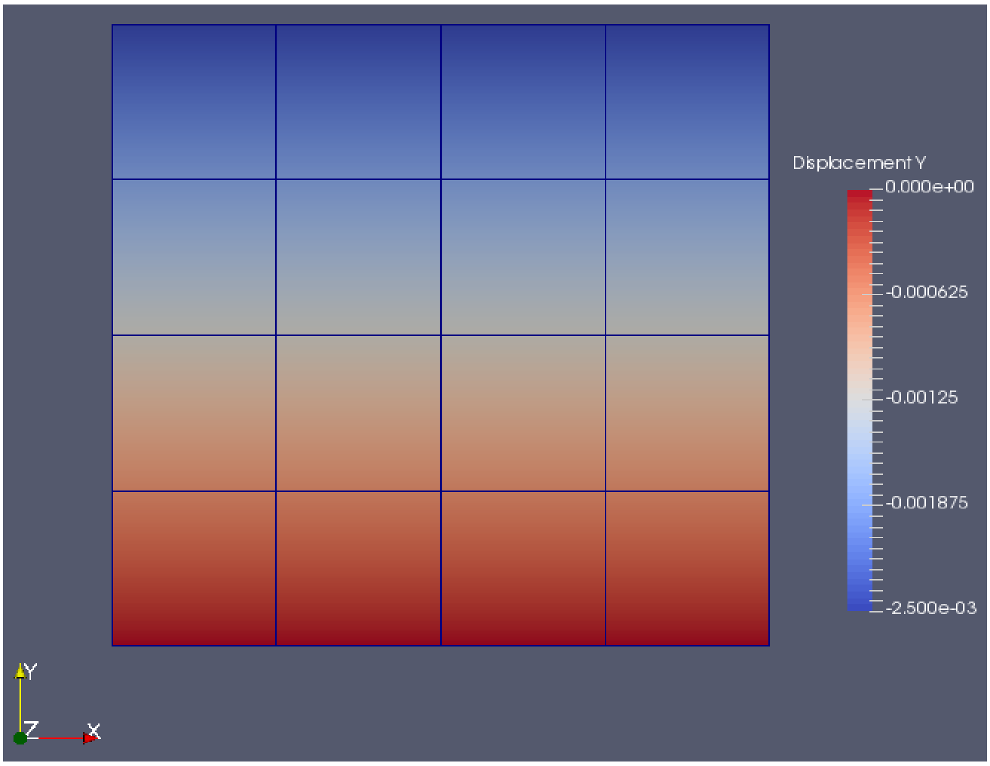

Figure 24: Deformation Y component for test 1 of linear quadrilateral elements.

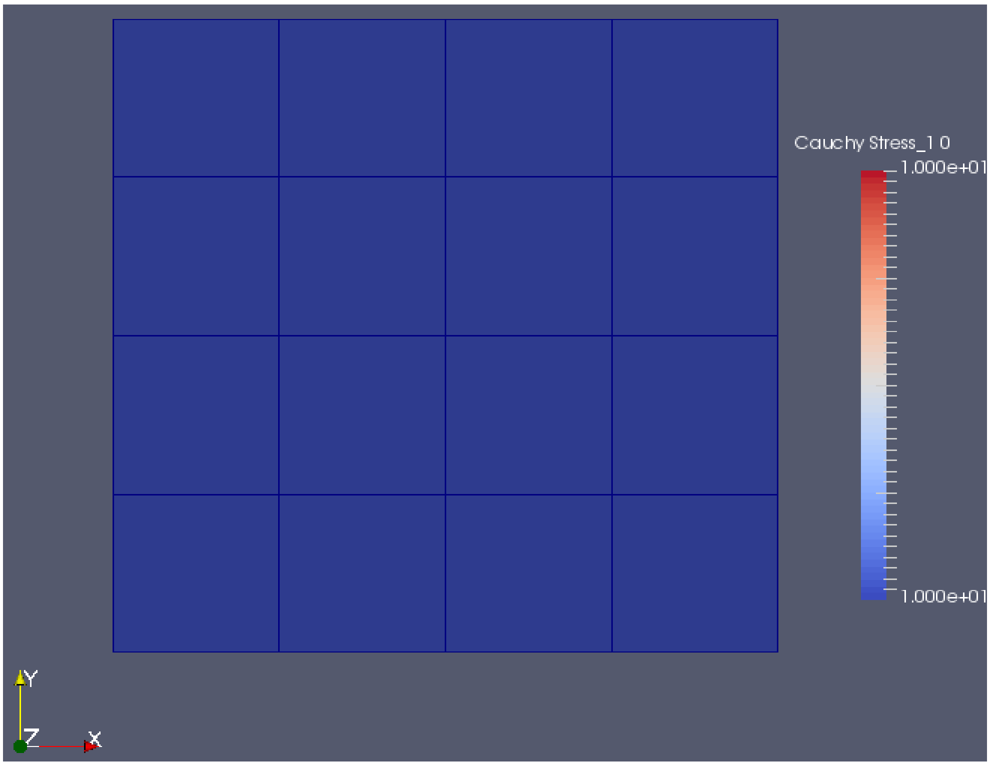

Figure 25: The xx component of the Cauchy stress for test 1. All other components were zero.

The displacement results for test 2 are shown in Figures 26 and 27. This displacement then caused the stress distribution shown in Figure 28.

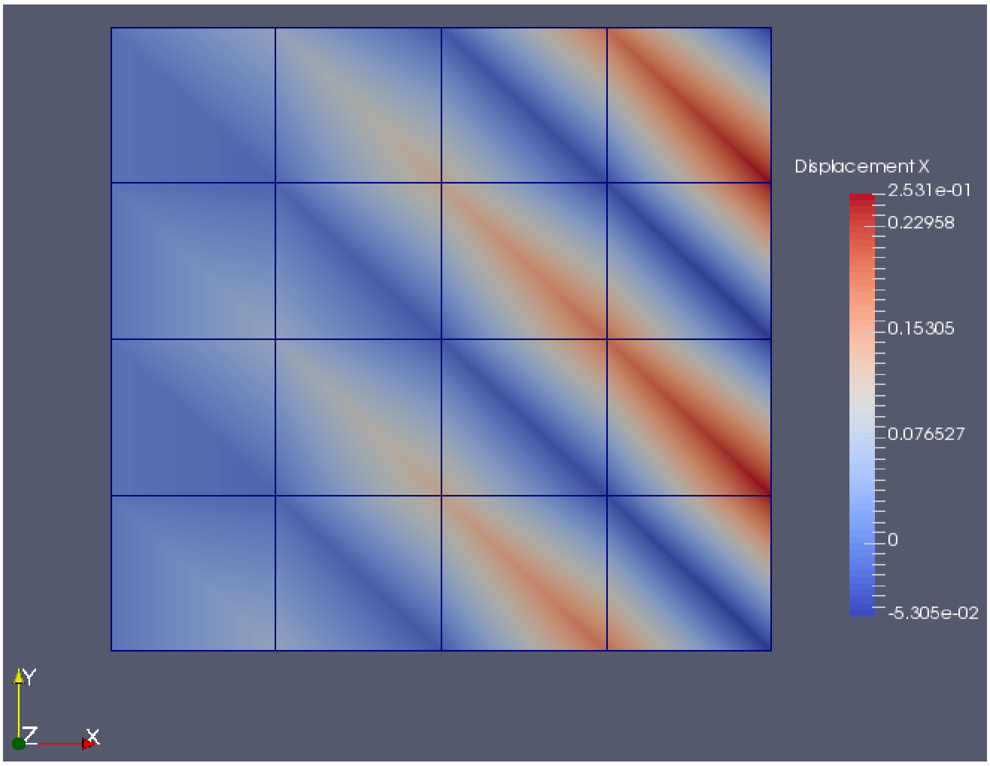

Figure 26: Deformation X component for test 2 of linear quadrilateral elements.

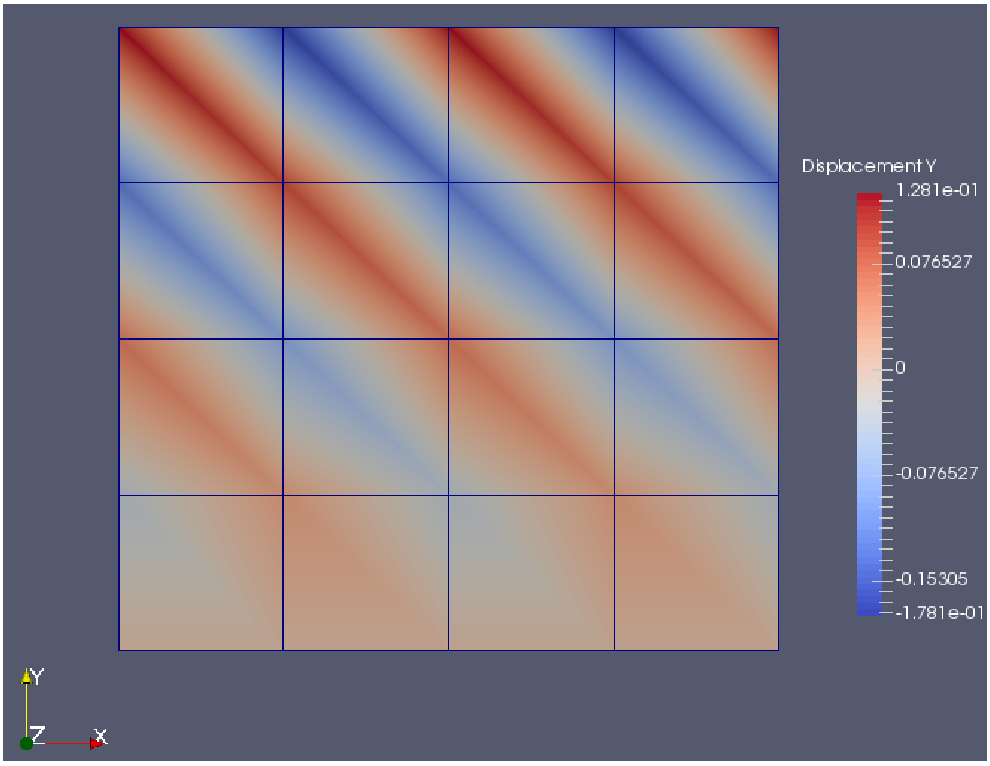

Figure 27: Deformation Y component for test 2 of linear quadrilateral elements.

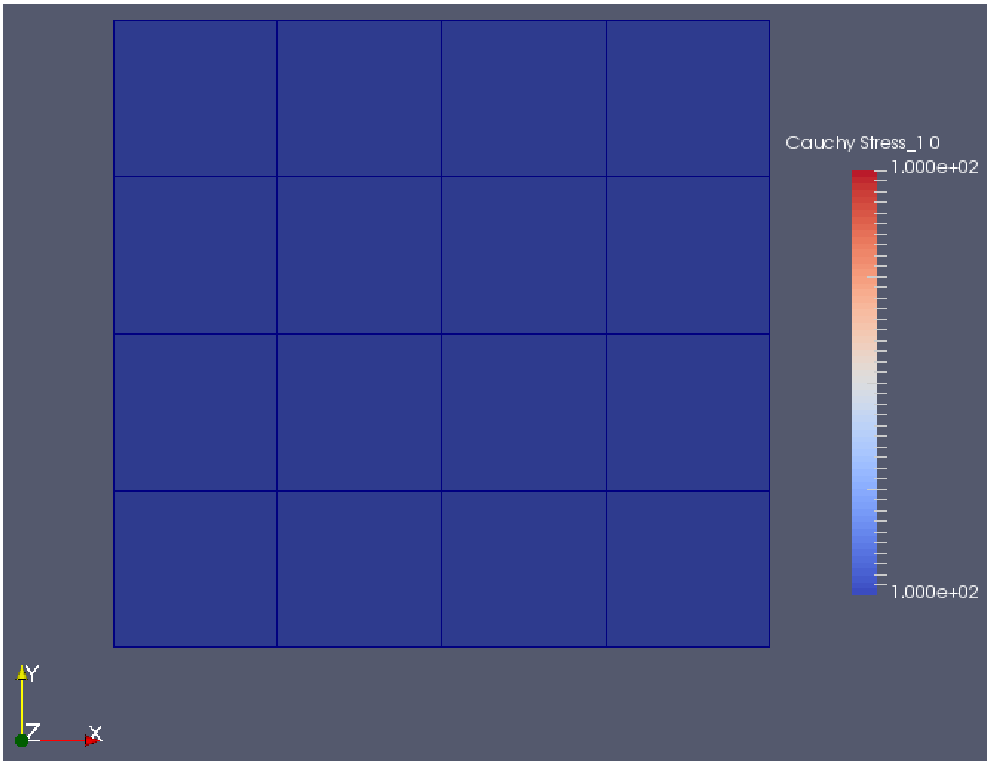

Figure 28: The xx component of the Cauchy stress for test 2. All other components were zero.

The displacement results for test 3 are shown in Figures 29 and 30. This displacement then caused the stress distribution shown in Figure 31.

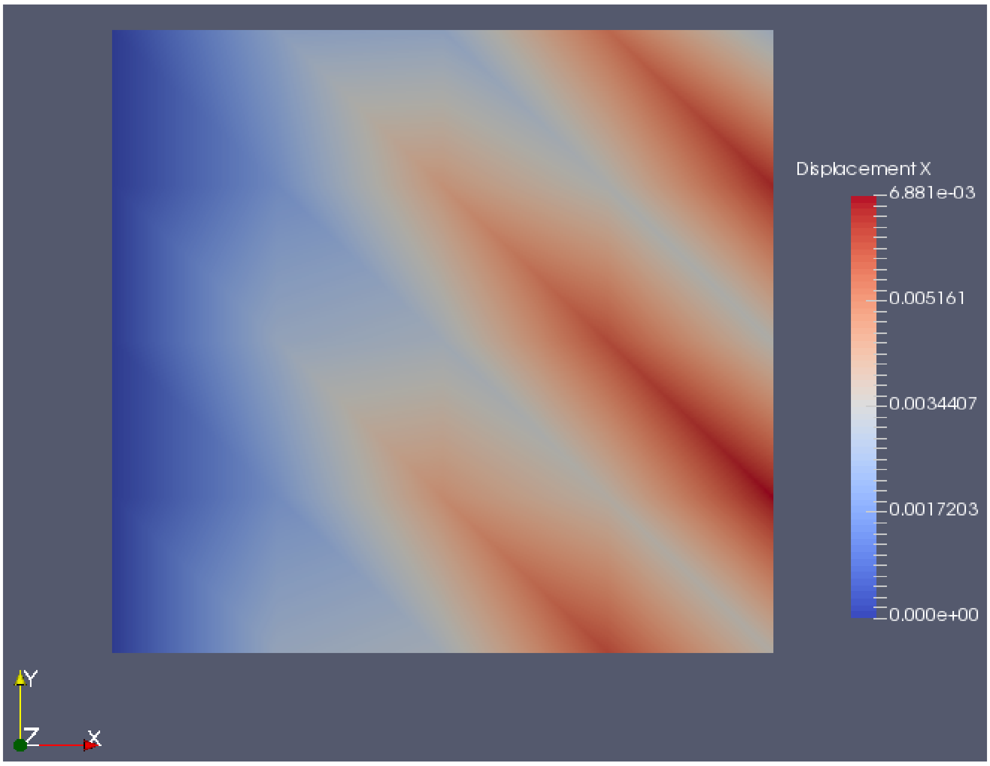

Figure 29: Deformation X component for test 3 of linear quadrilateral elements.

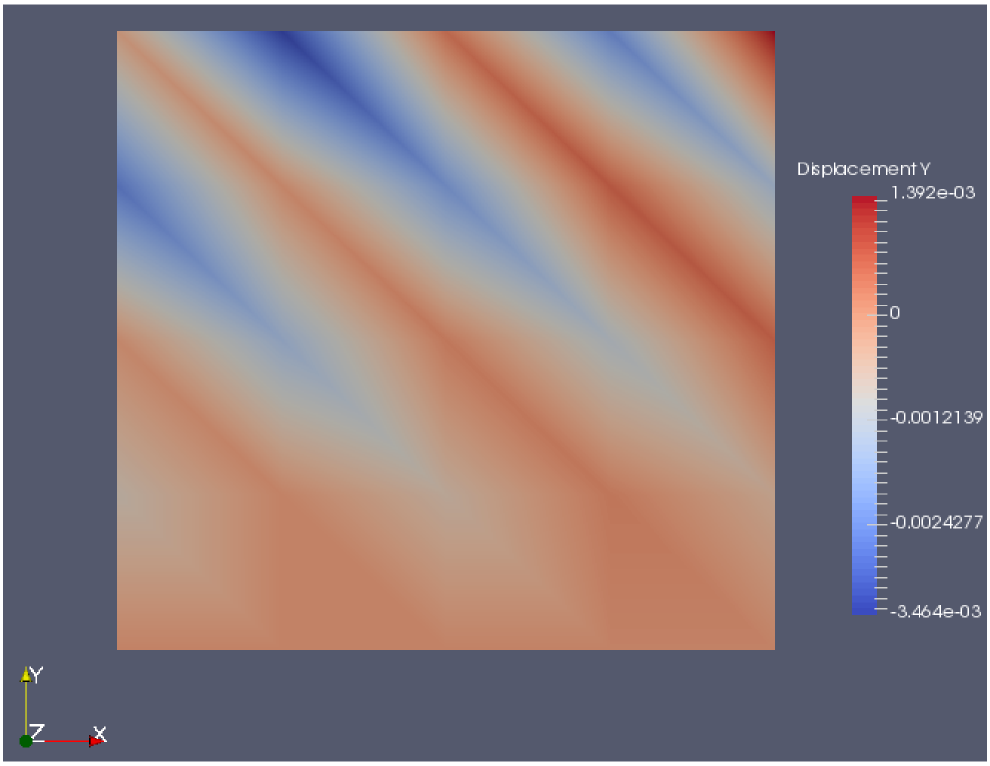

Figure 30: Deformation Y component for test 3 of linear quadrilateral elements.

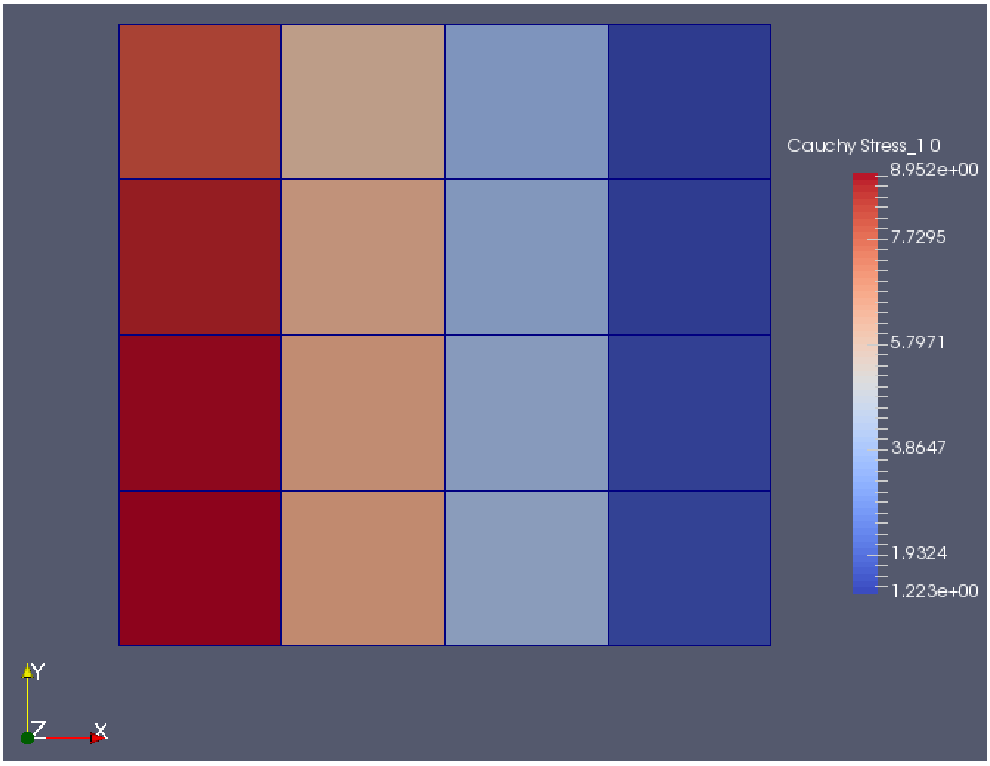

Figure 31: The xx component of the Cauchy stress for test 3.

4.4 Quadratic Quadrilaterals Elements

The displacement results for test 1 are shown in Figures 32 and 33. This displacement then caused the stress distribution shown in Figure 34. The original mesh used is shown in Figure 22.

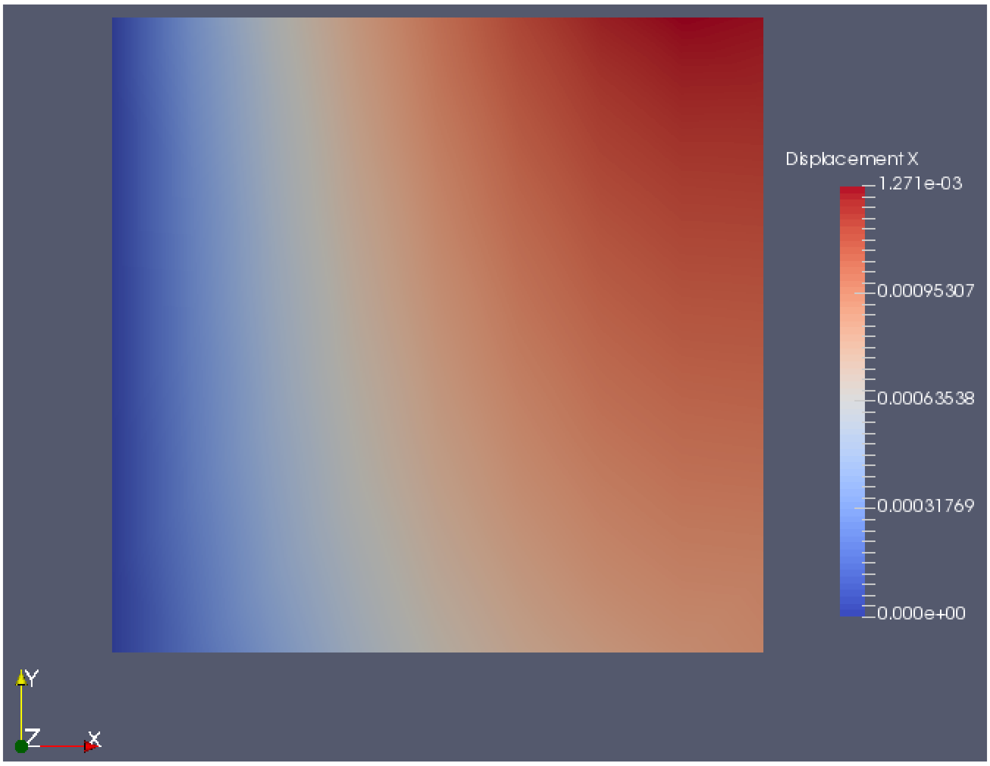

Figure 32: Deformation X component for test 1 of quadratic quadrilateral elements.

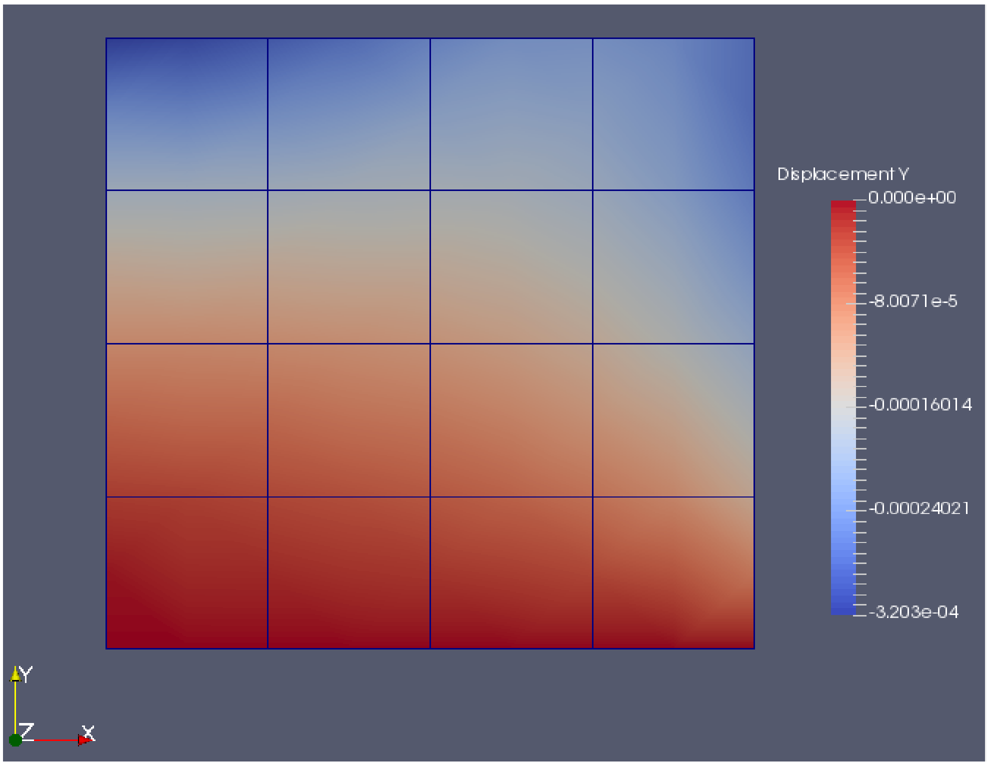

Figure 33: Deformation Y component for test 1 of quadratic quadrilateral elements.

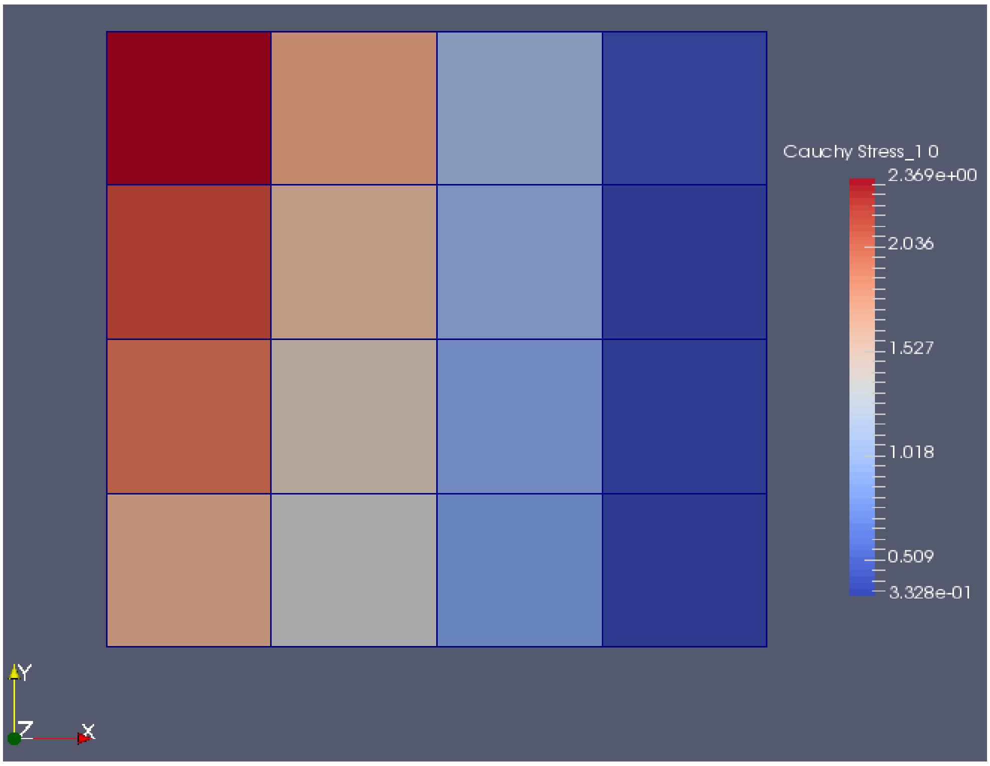

Figure 34: The xx component of the Cauchy stress for test 1. All other components were zero.

The displacement results for test 2 are shown in Figures 35 and 17. This displacement then caused the stress distribution shown in Figure 37.

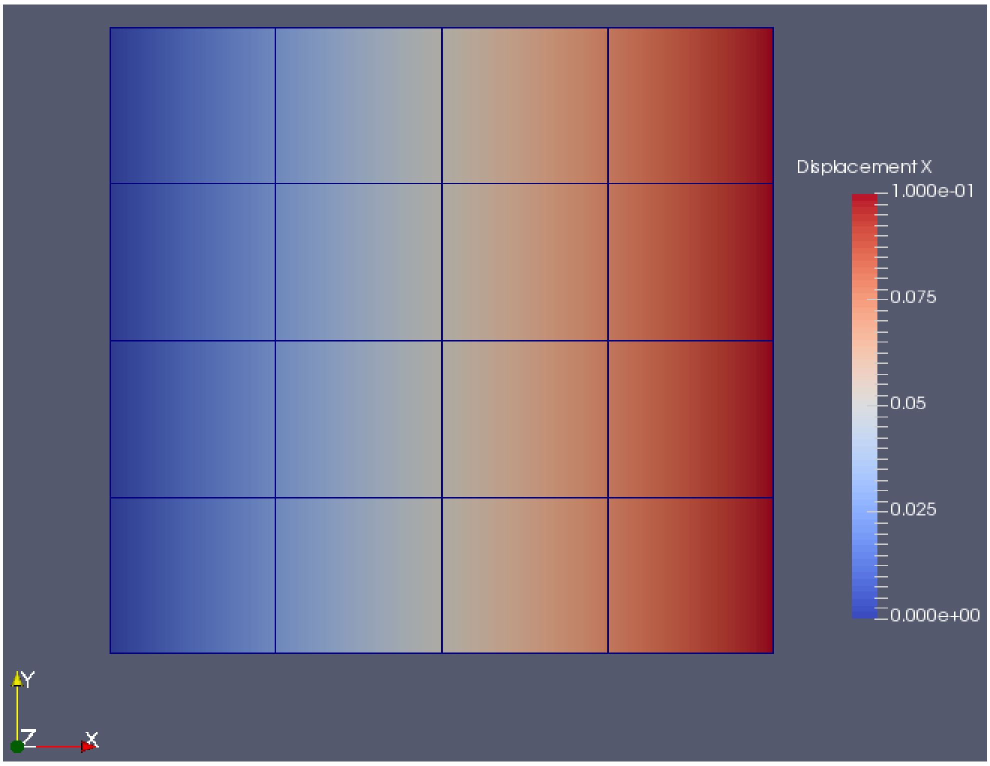

Figure 35: Deformation X component for test 2 of quadratic quadrilateral elements.

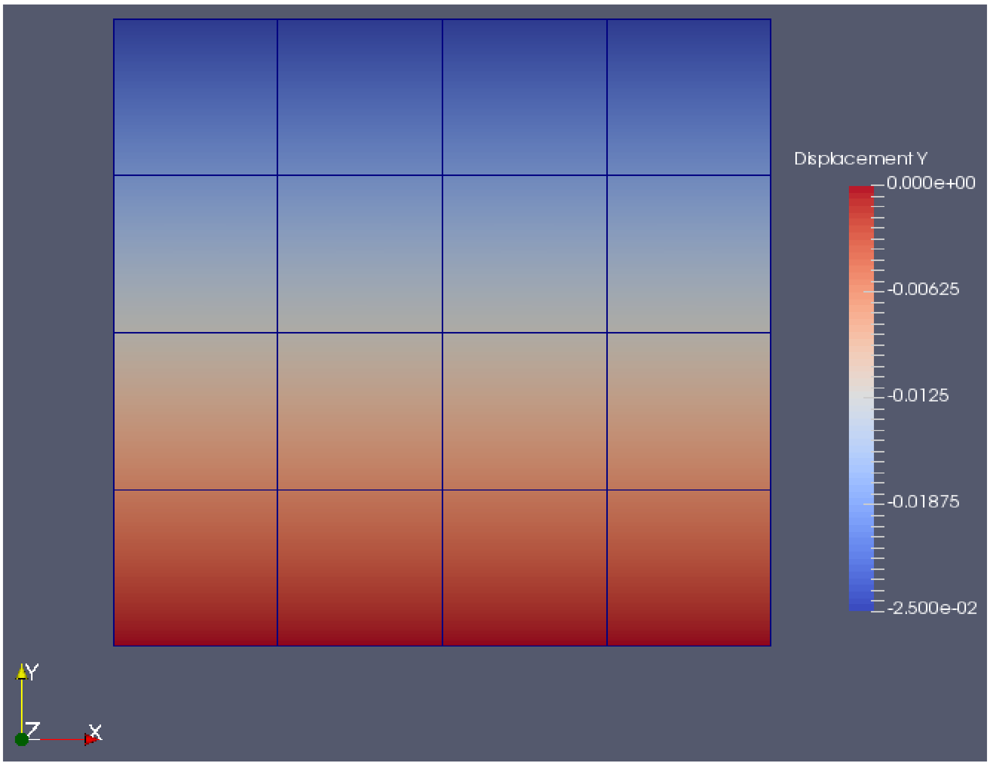

Figure 36: Deformation Y component for test 2 of quadratic quadrilateral elements.

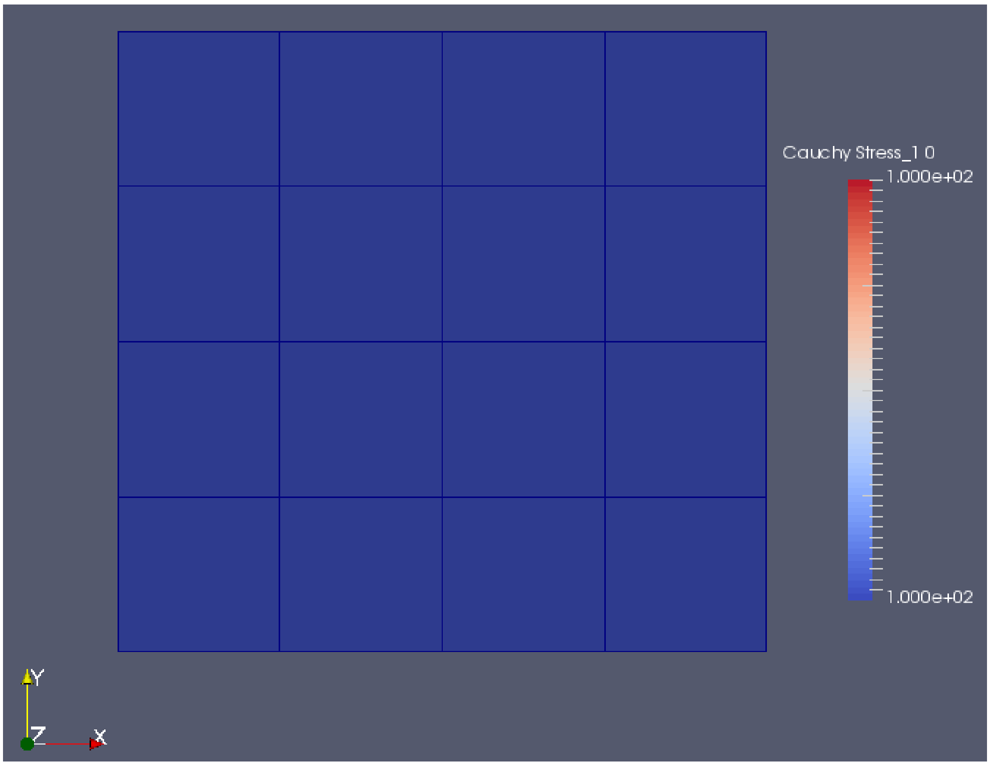

Figure 37: The xx component of the Cauchy stress for test 2. All other components were zero.

The displacement results for test 3 are shown in Figures 38 and 20. This displacement then caused the stress distribution shown in Figure 40.

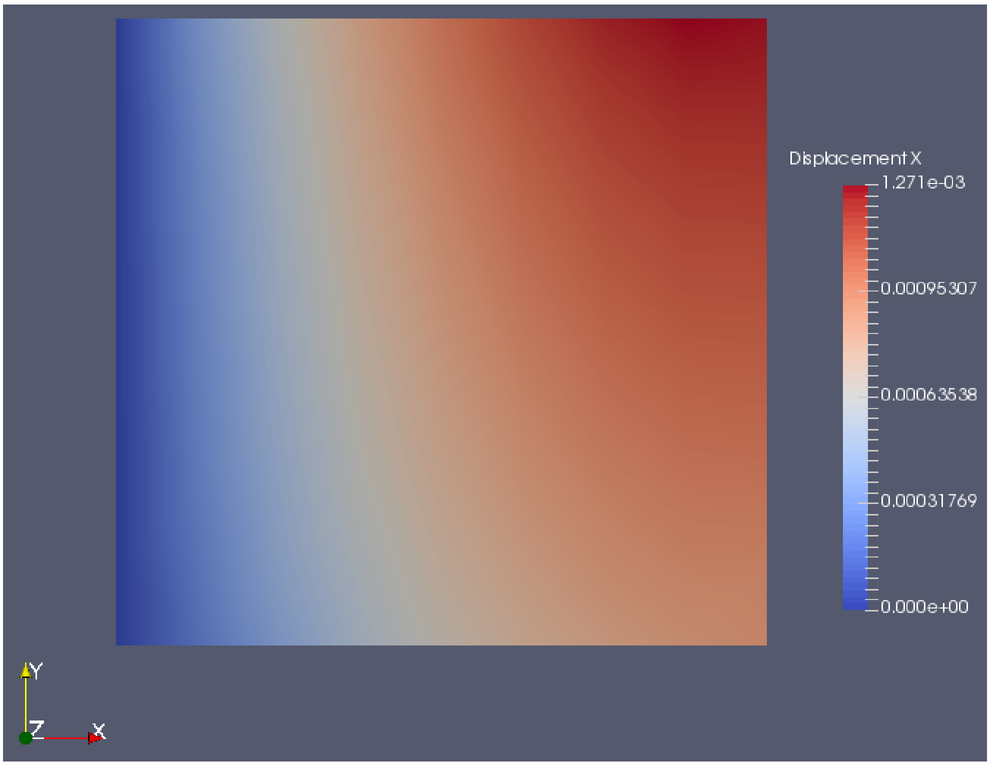

Figure 38: Deformation X component for test 3 of quadratic quadrilateral elements.

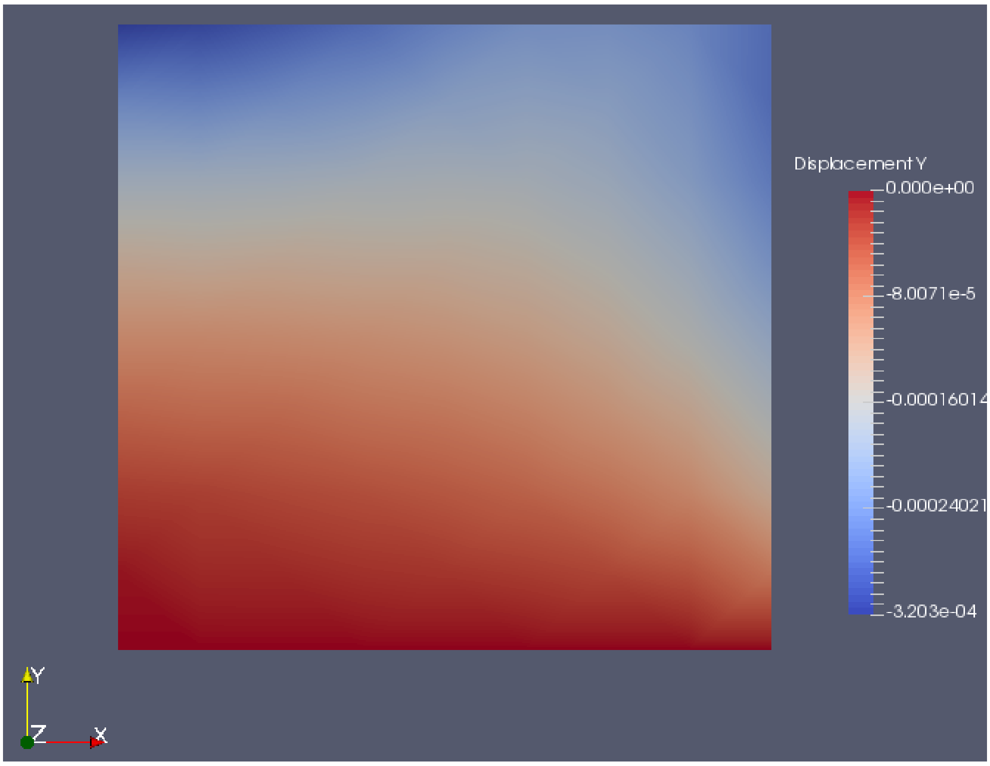

Figure 39: Deformation Y component for test 3 of quadratic quadrilateral elements.

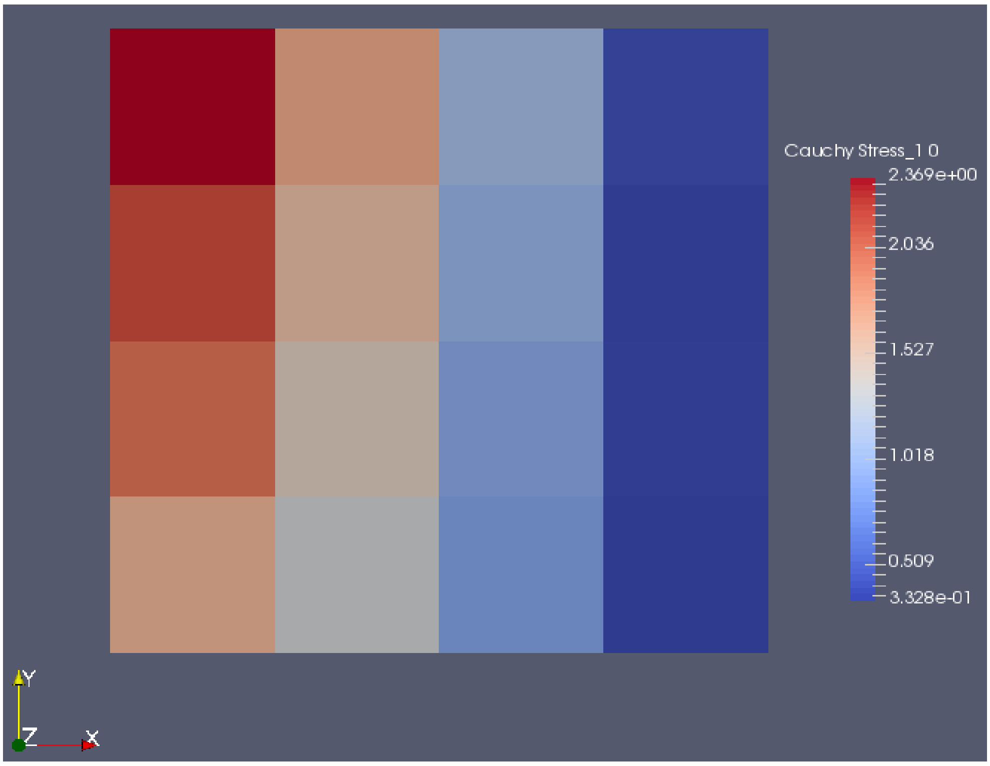

Figure 40: The xx component of the Cauchy stress for test 3.

5 Conclusion

The purpose of this project was to develop a FEA code that solves linear elliptic problems. This project did so for two dimensional plane stress solid mechanic problems.

The code presented would be able to solve similar two dimensional problems if needed. For example, plane strain problems could be evaluated by adjusting the material tensor D in the elemental stiffness integrator. The user provides the problem statement via geometric definition. This is shown by use of the `.yaml` and `associations` file; the user specifies collections of model entities on which boundary conditions are applied. The code is able to use either triangular or quadrilateral elements. It can also can use either first or second order shape functions and numerical integration schemes at the user's discretion. When building the linear system, the code uses the compressed row storage methods provided by the `Trilinos` software framework; the system is then solved using the GMRES method provided by the same framework. Once the solution, displacement components at the nodes, is recovered the secondary variable, stress, is also calculated.

Both of these quantities are then output graphically in the form of Vtk files.

5.1 Closing Comments

The displacement fields shown for the linear quadrilateral elements in Figures 26, 27, 29, and 30 are wrong as they do not make any physical sense. I wasn't able to figure out what was causing this; especially since the quadratic quadrilaterals worked as expected.

A better convergence study would have been to measure the stress magnification factor for a sufficiently large plate with a circular hole. This would have shown the convergence rates for each element combination independently as they approach the theoretical value of 3.

A Source Code and Headers

A.1 a4.cc

```
#include "A4_Disc.hpp"
#include "A4_Control.hpp"
#include "A4_FESolver.hpp"

// Trilinos Headers
#include <Teuchos_YamlParameterListHelpers.hpp>

// APF, GMI Headers
#include <apfMDS.h>
#include <apfMesh2.h>
#include <apf.h>
#include <apfShape.h>
#include <gmi_mesh.h>

#include <stdlib.h>

int main( int argc, char** argv)
{
  A4::initialize(true, true, true);

  Teuchos::ParameterList p;
  auto pp = Teuchos::Ptr<Teuchos::ParameterList> (&p);

  // Get info from argv's
  A4_ALWAYS_ASSERT(argc == 10);
  A4::print("USING MODEL: %s", argv[1]);
  A4::print("USING MESH: %s", argv[2]);
  A4::print("USING ASSOC: %s", argv[3]);
  A4::print("SOLVING: %s", argv[5]);
  A4::print("INTEGRATION ORDER: %s", argv[6]);
  A4::print("BODY LOAD: %s, %s, %s", argv[7], argv[8], argv[9]);
  double bodyLoad[3] = {
    std::atof( argv[7]),
    std::atof( argv[8]),
    std::atof( argv[9])};
  int order = std::atoi( argv[6]);
```

```
  Teuchos::updateParametersFromYamlFile(argv[4], pp);

  // Load mesh and create discretization object
  gmi_register_mesh();
  apf::Mesh2* m = apf::loadMdsMesh(argv[1], argv[2]);

  if( order == 1)
  {
    // No changes
  }
  else if( order == 2)
  {
    auto shape = apf::getSerendipity();
    m->changeShape( shape);
  }
  else
  {
    std::cout << "Order" << order << " not supported" << std::endl;
    std::abort();
  }

  auto disc = new A4::Disc(m, argv[3]);

  // Create linear algebra solver and solve system
  auto solver = new A4::FESolver(disc, *pp, order, bodyLoad);
  solver->solve();

  // Write out solution, force, and stress to fields as vtk files
  char disp_name[] = "Displacement";
  apf::Field* disp = apf::createFieldOn( m, disp_name, apf::VECTOR);
  apf::zeroField( disp);
  solver->set_disp_to_field( disp);

  char force_name[] = "Traction";
  apf::Field* force = apf::createFieldOn( m, force_name, apf::VECTOR);
```

```
  apf::zeroField( force);
  solver->set_force_to_field( force);

  char stress_name[] = "Cauchy Stress";
  apf::Field* stress_f = apf::createIPField(
      m, stress_name, apf::MATRIX, (m->getShape())->getOrder());
  apf::zeroField( stress_f);
  solver->set_stress_to_field( stress_f);

  // Write solution
  char* fileName = argv[5];
  apf::writeVtkFiles( fileName, m, m->getDimension());

  std::cout << "APPROXIMATION ERROR IS: " << solver->get_error() << std::endl;

  // delete fields
  apf::destroyField( disp);
  apf::destroyField( force);
  apf::destroyField( stress_f);

  // delete discretization and solver objects
  delete solver;
  delete disc;

  // destroy the mesh
  m->destroyNative();
  apf::destroyMesh(m);

  // close mpi services
  A4::finalize();

  return 0;
}
```

A.2 A4_BodyLoads.hpp

```
#ifndef A4_BODYLOADS_HPP
#define A4_BODYLOADS_HPP

/// @file A4_BodyLoads.hpp

#include <A4_Disc.hpp>
#include <A4_LinAlg.hpp>

#include <apf.h>
#include <apfShape.h>
#include <apfDynamicMatrix.h>

namespace Teuchos
{
class ParameterList;
} // End namespace Teuchos

namespace A4
{

class LinAlg;

/// @brief Precribe body loads.
/// @param d  The discretization of node sets.
/// @param la The relavent linear algebra data.
/// @param order
///            The order of numerical integration accuracy.
/// @param g  The value of the body load.
void apply_body_loads(
  Disc* d,
  LinAlg* la,
  int order,
  double* g);

/// @brief Class to compute elemental force vector due to body load.
class BodyLoad :
  public apf::Integrator
```

```
{
 public:

   /// @brief Construct elemental body force integrator
   /// @param mesh  The mesh to work with.
   /// @param order The order of numerical integration.
   /// @param load  The body load vector.
   BodyLoad(
       apf::Mesh* mesh,
       int inte_order,
       apf::Vector3 load);


   /// @brief Prepares each new element.
   /// elem The new element to be processed.
   void inElement(
       apf::MeshElement* me);

   /// @brief Work done at each integration point.
   /// @param p  The parametric integration point coordinates.
   /// @param w  The integration point weight.
   /// @param dv The differential volume at the integration point.A
   void atPoint(
       apf::Vector3 const& p,
       double w,
       double dv);

   /// @brief Free and finalize data once done with the element.
   void outElement();

   /// @brief Get the elemental forcing vector due only to body loads.
   apf::DynamicVector get_fe()
   {
     return fe;
   }

```

```
  private:
    int order;
    apf::Mesh* m;
    apf::FieldShape* shape;
    apf::MeshElement* elem;

    apf::Vector3 g;
    apf::NewArray<double> N;
    apf::DynamicVector fe;

    int num_dims;
    int num_elem_nodes;
    int num_elem_dofs;

};

} // end namspace A4
#endif
```

A.3 A4_BodyLoads.cpp

```
#include <A4_BodyLoads.hpp>

namespace A4
{

using Teuchos::getValue;
using Teuchos::Array;

void apply_body_loads(
  Disc* d,
  LinAlg* la,
  int order,
  double load[3])
{
  // Get info from inputs
  apf::Vector3 bodyLoad ( load[0], load[1], load[2]);
  auto mesh = d->get_apf_mesh();

  auto blInter = new BodyLoad( mesh, order, bodyLoad);

  // Iterate over all highest dimension mesh elements
  apf::MeshEntity* ent;
  auto it = mesh->begin( mesh->getDimension());
  while( (ent = mesh->iterate(it)) )
  {
    // Get local ids of the dofs of this entity
    std::vector<LO> lids;
    d->get_ghost_lids( ent, lids);

    apf::MeshElement* elem = apf::createMeshElement( mesh, ent);

    // Integrate over this element
    blInter->process( elem);
    auto fe = blInter->get_fe();

    // Sum into global forcing vector
    for( size_t i = 0; i < fe.size(); i++)
    {
```

```
      LO row = lids[i];
      auto f = fe[i];
      la->ghost->F->sumIntoLocalValue( row, f);
    }

    apf::destroyMeshElement( elem);
  }
  mesh->end( it);

  delete blInter;

  return;
}

BodyLoad::BodyLoad(
    apf::Mesh* mesh,
    int inte_order,
    apf::Vector3 load):
  apf::Integrator( inte_order)
{
  m = mesh;
  order = inte_order;
  shape = m->getShape();
  g = load;
  return;
}

void BodyLoad::inElement(
    apf::MeshElement* me)
{
  elem = me;
  auto ent = apf::getMeshEntity( elem);
  auto type = m->getType( ent);
  auto es = shape->getEntityShape( type);
  num_dims = m->getDimension();
  num_elem_nodes = es->countNodes();
  num_elem_dofs = num_dims * num_elem_nodes;

  fe.setSize( num_elem_dofs);
  fe.zero();
```

```

  return;
}

void BodyLoad::atPoint(
        apf::Vector3 const& p,
        double w,
        double dv)
{
  apf::getBF( shape, elem, p, N);
  apf::DynamicVector f_tmp (num_elem_dofs);
  f_tmp.zero();
  fe.zero();
  int ind = 0;

  for( int i =0; i < num_elem_nodes; ++i)
  {
    for( int j = 0; j < num_dims; ++j)
    {
      ind = i*num_dims+j;
      f_tmp[ind] += N[i]*g[j]*w*dv;
    }
  }

  fe += f_tmp;
  return;
}

void BodyLoad::outElement()
{
  elem = 0;
  return;
}

} // End namespace A4
```

A.4 A4_Control.hpp

```cpp
#ifndef A4_CONTROL_HPP
#define A4_CONTROL_HPP

/// @file A4_Control.hpp

#include <string>

namespace A4 {

/// @brief Initialize parallel and expression parsing services.
/// @param init_mpi Should a call be made to initialize MPI?
/// @param init_kokkos Should a call be made to initialize Kokkos?
/// @param init_pcu Should a call be made to initialize PCU?
/// @details This method calls initialize routines for MPI and PCU,
/// as well as initializing a real-time string expression parser.
void initialize(
    bool init_mpi = true,
    bool init_kokkos = true,
    bool init_pcu = true);

/// @brief Finalize the parallel services.
/// @details This method calls finalize routines for MPI and PCU
/// if they were initialized with A4::initialize.
void finalize();

/// @brief Print a printf-style formatted message on rank 0.
void print(const char* msg, ...);

/// @brief Fail the application with an explanation message.
```

```
void fail(const char* why, ...) __attribute__((noreturn));

/// @brief Fail the application by assertion, with an error message.
void assert_fail(const char* why, ...) __attribute__((noreturn));

/// @brief Evaluate a string expression.
/// @param v The input mathematical string expression.
/// @param x The x coordinate location.
/// @param y The y coordinate location.
/// @param z The z coordinate location.
/// @param t The current time.
double eval(
    std::string const& v,
    const double x,
    const double y,
    const double z,
    const double t);

/// @brief Get the wall time in seconds.
double time();

} // end namespace A4

/// @brief Always assert a conditional
#define A4_ALWAYS_ASSERT(cond)                              \
  do {                                                      \
    if (! (cond)) {                                         \
      char omsg[2048];                                      \
      sprintf(omsg, "%s failed at %s + %d \n",              \
        #cond, __FILE__, __LINE__);                         \
      A4::assert_fail(omsg);                              \
    }                                                       \
  } while (0)

/// @brief Always assert a conditional with a message
#define A4_ALWAYS_ASSERT_VERBOSE(cond, msg)                 \
  do {                                                      \
```

```
    if (! (cond)) {                                         \
      char omsg[2048];                                      \
      sprintf(omsg, "%s failed at %s + %d \n%s \n",         \
        #cond, __FILE__, __LINE__, msg);                    \
      A4::assert_fail(omsg);                                \
    }                                                       \
  } while(0)

#ifdef NDEBUG
/// @brief Do nothing - optimized out
#define A4_DEBUG_ASSERT(cond)
/// @brief Do nothing - optimized out
#define A4_DEBUG_ASSERT_VERBOSE(cond, msg)
#else
/// @brief Assert a conditional
#define A4_DEBUG_ASSERT(cond) \
  A4_ALWAYS_ASSERT(cond)
/// @brief Assert a conditional
#define A4_DEBUG_ASSERT_VERBOSE(cond, msg) \
  A4_ALWAYS_ASSERT_VERBOSE(cond, msg)
#endif

#endif
```

A.5 A4_Control.cpp

```
#include <cstdarg>
#include <cstdlib>
#include <PCU.h>
#include <Kokkos_Core.hpp>

#include "A4_Control.hpp"

namespace A4 {

static bool is_goal_initd = false;
static bool is_mpi_initd = false;
static bool is_kokkos_initd = false;
static bool is_pcu_initd = false;

static void call_mpi_init()
{
  MPI_Init(0, 0);
  is_mpi_initd = true;
}

static void call_pcu_init()
{
  PCU_Comm_Init();
  is_pcu_initd = true;
}

static void call_kokkos_init()
{
  Kokkos::initialize();
  is_kokkos_initd = true;
}

void initialize(bool init_mpi, bool init_kokkos, bool init_pcu)
{
  if (is_goal_initd) return;
  if (init_mpi) call_mpi_init();
  if (init_kokkos) call_kokkos_init();
  if (init_pcu) call_pcu_init();
```

```
  is_goal_initd = true;
}

static void call_mpi_free()
{
  MPI_Finalize();
  is_mpi_initd = false;
}

static void call_kokkos_free()
{
  Kokkos::finalize();
  is_kokkos_initd = false;
}

static void call_pcu_free()
{
  PCU_Comm_Free();
  is_pcu_initd = false;
}

void finalize()
{
  if (is_pcu_initd) call_pcu_free();
  if (is_kokkos_initd) call_kokkos_free();
  if (is_mpi_initd) call_mpi_free();
  is_goal_initd = false;
}

void print(const char* message, ...)
{
  if (PCU_Comm_Self()) return void();
  va_list ap;
  va_start(ap, message);
  vfprintf(stdout, message, ap);
  va_end(ap);
  printf("\n");
}

void fail(const char* why, ...)
```

```
{
  va_list ap;
  va_start(ap, why);
  vfprintf(stderr, why, ap);
  va_end(ap);
  printf("\n");
  abort();
}

void assert_fail(const char* why, ...)
{
  fprintf(stderr, "%s", why);
  abort();
}

double time()
{
  return PCU_Time();
}

} // end namespace A4
```

A.6 A4_DBCs.hpp

```
#ifndef A4_DBCS_HPP
#define A4_DBCS_HPP

/// @file A4_DBCs.hpp

#include "A4_Disc.hpp"

namespace Teuchos {
class ParameterList;
}

namespace A4 {

using Teuchos::ParameterList;

class LinAlg;

/// @brief Prescribe Dirichlet boundary conditions.
/// @param d  The discretization of node sets.
/// @param la The relevant linear algebra data.
/// @param p  The parameterlist defining bcs.
void apply_dbcs(
    Disc* d,
    LinAlg* la,
    ParameterList const& p);

}

#endif
```

A.7 A4_DBCs.cpp

```cpp
#include "A4_DBCs.hpp"
#include "A4_Disc.hpp"
#include "A4_LinAlg.hpp"

#include <Teuchos_ParameterList.hpp>
#include <string>

namespace A4 {

using Teuchos::getValue;
using Teuchos::Array;

void apply_dbcs(
    Disc* d,
    LinAlg* la,
    ParameterList const& p)
{
  auto sublist = p.sublist("dirichlet_bcs");
  std::vector<apf::Node> nodes;
  auto K = la->owned->K;
  auto f = la->owned->F->get1dViewNonConst();
  Array<LO> col_indices;
  Array<ST> col_entries;
  ST diag_entry = 1.0;
  size_t num_entries;

  for( auto it = sublist.begin(); it != sublist.end(); ++it)
  {
    auto entry = sublist.entry(it);
    // Format is: {0, xmin, 0.0} (spatial_dof, node_set_name, value)
    auto a = getValue<Array<std::string> >(entry);
    int s_dof = std::stoi(a[0]);
    std::string nodeSet = a[1];
    double value = std::stod(a[2]);

    nodes = d->get_nodes( nodeSet);
    for( int i = 0; i<(int)nodes.size(); ++i)
```

```
    {
      auto node = nodes[i];
      auto row = d->get_owned_lid( node, s_dof);

      num_entries = K->getNumEntriesInLocalRow(row);
      col_indices.resize(num_entries);
      col_entries.resize(num_entries);
      K->getLocalRowCopy(row, col_indices(), col_entries(), num_entries);

      for (size_t c = 0; c < num_entries; ++c)
      {
        if( !(col_indices[c] == row))
        {
          col_entries[c] = 0.0;
        }
        else
        {
          diag_entry = col_entries[c];
        }
      }
      K->replaceLocalValues(row, col_indices(), col_entries());
      f[row] = diag_entry*value;
    }
  }
  return;
}

}
```

A.8 A4_Disc.hpp

```cpp
#ifndef A4_DISC_HPP
#define A4_DISC_HPP

/// @file A4_Disc.hpp

#include <apfNumbering.h>
#include "A4_Defines.hpp"

/// @cond
// forward declarations
namespace apf {
struct Node;
struct StkModels;
class Mesh;
class MeshEntity;
}
/// @endcond

namespace A4 {

/// @brief A discretization container used to:
/// - construct Tpetra maps and graphs that are used to build
///   linear algebra objects.
/// - gather mesh entities associated with Dirichlet and Neumann
///   boundary conditions, as defined in an input model associations
///   file.
class Disc {

  public:

    /// @brief Construct the discretization object.
    /// @param m The input mesh object.
    /// @param assoc The input model association file name.
    Disc(apf::Mesh* m, const char* assoc);

```

```
    /// @brief Destroy the discretization object.
    /// @details This will destroy the association objects, and the Tpetra
    /// linear algebra maps and graphs that this object creates.
    ~Disc();

    /// @brief Returns the underlying AA4 mesh.
    apf::Mesh* get_apf_mesh() { return mesh; }

    /// @brief Returns the model association definitions.
    apf::StkModels* get_model_sets() { return sets; }

    /// @brief Returns the owned DOF map.
    RCP<const Map> get_owned_map() { return owned_map; }

    /// @brief Returns the ghost DOF map.
    RCP<const Map> get_ghost_map() { return ghost_map; }

    /// @brief Returns the owned element connectivity graph.
    RCP<const Graph> get_owned_graph() { return owned_graph; }

    /// @brief Returns the ghost element connectivity graph.
    RCP<const Graph> get_ghost_graph() { return ghost_graph; }

    /// @brief Returns the nodal coordinates.
    RCP<MultiVector> get_coords() { return coords; }

    /// @brief Returns the total number of entity DOFs.
    int get_num_dofs(apf::MeshEntity* e);

    apf::Numbering* get_owned_numbering() {return owned_nmbr;}

    /// @brief Returns the local row IDs of DOFs on a mesh entity.
```

```
    void get_ghost_lids(apf::MeshEntity* e, std::vector<LO>& lids);

    /// @brief Returns the local row ID of a node
    LO get_owned_lid(apf::Node const& n, int eq);

    /// @brief Returns the number of side sets.
    int get_num_side_sets() const;

    /// @brief Returns the number of node sets.
    int get_num_node_sets() const;

    /// @brief Returns the name of the ith side set.
    std::string get_side_set_name(const int i) const;

    /// @brief Returns the name of the ith node set.
    std::string get_node_set_name(const int i) const;

    /// @brief Returns the sides in a side set.
    /// @param set The name of the side set of interest.
    std::vector<apf::MeshEntity*> const& get_sides(std::string const& set);

    /// @brief Returns the nodes in a node set.
    /// @param set The name of the node set of interest.
    std::vector<apf::Node> const& get_nodes(std::string const& set);

  private:

    void compute_owned_map();
    void compute_ghost_map();
    void compute_graphs();
    void compute_coords();
    void compute_side_sets();
    void compute_node_sets();

    int num_eqs;
    int num_dims;

```

```
    apf::Mesh* mesh;
    apf::StkModels* sets;
    apf::FieldShape* shape;
    apf::Numbering* owned_nmbr;
    apf::Numbering* ghost_nmbr;
    apf::GlobalNumbering* global_nmbr;
    apf::DynamicArray<apf::Node> owned;

    RCP<const Comm> comm;
    RCP<const Map> owned_map;
    RCP<const Map> ghost_map;
    RCP<const Map> node_map;
    RCP<MultiVector> coords;
    RCP<Graph> owned_graph;
    RCP<Graph> ghost_graph;

    std::map<std::string, std::vector<apf::Node> > ↩
       node_sets;
    std::map<std::string, std::vector<apf::MeshEntity*> >↩
        side_sets;
};

} // end namespace A4

#endif
```

A.9 A4_Disc.cpp

```cpp
#include <apf.h>
#include <apf.h>
#include <apfAlbany.h>
#include <apfMDS.h>
#include <apfMesh2.h>
#include <apfShape.h>
#include <gmi_mesh.h>

#include "A4_Disc.hpp"
#include "A4_Control.hpp"

namespace A4 {

static apf::StkModels* read_sets(apf::Mesh* m, const char* filename)
{
  auto sets = new apf::StkModels;
  print("reading association file: %s", filename);
  static std::string const setNames[3] = {
      "node set", "side set", "elem set"};
  auto d = m->getDimension();
  int dims[3] = {0, d - 1, d};
  std::ifstream f(filename);
  if (!f.good()) fail("cannot open file: %s", filename);
  std::string sline;
  int lc = 0;
  while (std::getline(f, sline)) {
    if (!sline.length()) break;
    ++lc;
    int sdi = -1;
    for (int di = 0; di < 3; ++di)
      if (sline.compare(0, setNames[di].length(), setNames[di]) == 0) sdi = di;
    if (sdi == -1)
      fail("invalid association line # %d:\n\t%s", lc, sline.c_str());
    int sd = dims[sdi];
    std::stringstream strs(sline.substr(setNames[sdi].length()));
```

```
    auto set = new apf::StkModel();
    strs >> set->stkName;
    int nents;
    strs >> nents;
    if (!strs) fail("invalid association line #%d:\n\t%s", lc, sline.c_str());
    for (int ei = 0; ei < nents; ++ei) {
      std::string eline;
      std::getline(f, eline);
      if (!f || !eline.length())
        fail("invalid association after line #%d", lc);
      ++lc;
      std::stringstream strs2(eline);
      int mdim, mtag;
      strs2 >> mdim >> mtag;
      if (!strs2) fail("bad associations line #%d:\n\t%s", lc, eline.c_str());
      set->ents.push_back(m->findModelEntity(mdim, mtag));
      if (!set->ents.back())
        fail("no model entity with dim: %d and tag: %d", mdim, mtag);
    }
    sets->models[sd].push_back(set);
  }
  sets->computeInverse();
  return sets;
}

Disc::Disc(apf::Mesh* m, const char* assoc)
    : mesh(m),
      shape(m->getShape()),
      owned_nmbr(0),
      ghost_nmbr(0),
      global_nmbr(0)
{
  auto t0 = time();
  mesh->verify();
  sets = read_sets(m, assoc);
  num_eqs = mesh->getDimension();
```

```
  num_dims = mesh->getDimension();
  comm = Tpetra::DefaultPlatform::getDefaultPlatform().getComm();
  compute_owned_map();
  compute_ghost_map();
  compute_graphs();
  compute_coords();
  compute_side_sets();
  compute_node_sets();
  auto t1 = time();
  print(" num side sets: %d", get_num_side_sets());
  print(" num node sets: %d", get_num_node_sets());
  print(" disc built in %f seconds", t1 - t0);
}

Disc::~Disc()
{
  if (sets) delete sets;
  comm = Teuchos::null;
  owned_map = Teuchos::null;
  owned_map = Teuchos::null;
  ghost_map = Teuchos::null;
  node_map = Teuchos::null;
  coords = Teuchos::null;
  owned_graph = Teuchos::null;
  ghost_graph = Teuchos::null;
}

static LO get_dof(const LO nid, const int eq, const int eqs)
{
  return nid * eqs + eq;
}

static GO get_gdof(const GO nid, const int eq, const int eqs)
{
  return nid * eqs + eq;
}

```

```
int Disc::get_num_dofs(apf::MeshEntity* e) {
  auto type = mesh->getType(e);
  auto es = shape->getEntityShape(type);
  auto num_nodes = es->countNodes();
  return num_nodes * num_eqs;
}

void Disc::get_ghost_lids(apf::MeshEntity* e, std::vector<LO>& lids) {
  lids.resize(get_num_dofs(e));
  static apf::NewArray<int> node_ids;
  int num_nodes = apf::getElementNumbers(ghost_nmbr, e, node_ids);
  int dof = 0;
  for (int node = 0; node < num_nodes; ++node)
    for (int eq = 0; eq < num_eqs; ++eq)
      lids[dof++] = get_dof(node_ids[node], eq, num_eqs);
}

LO Disc::get_owned_lid(apf::Node const& n, int eq)
{
  LO nid = apf::getNumber(owned_nmbr, n.entity, n.node, 0);
  return get_dof(nid, eq, num_eqs);
}

int Disc::get_num_side_sets() const
{
  return sets->models[num_dims - 1].size();
}

int Disc::get_num_node_sets() const
{
  return sets->models[0].size();
}

std::string Disc::get_side_set_name(const int i) const
{
  A4_DEBUG_ASSERT(i < get_num_side_sets());
  return sets->models[num_dims - 1][i]->stkName;
```

```
}

std::string Disc::get_node_set_name(const int i) const
{
  A4_DEBUG_ASSERT(i < get_num_node_sets());
  return sets->models[0][i]->stkName;
}

std::vector<apf::MeshEntity*> const& Disc::get_sides(
    std::string const& side_set)
{
  if (! side_sets.count(side_set))
    fail("side set %s not found", side_set.c_str());
  return side_sets[side_set];
}

std::vector<apf::Node> const& Disc::get_nodes(
    std::string const& node_set)
{
  if (! node_sets.count(node_set))
    fail("node set %s not found", node_set.c_str());
  return node_sets[node_set];
}

void Disc::compute_owned_map()
{
  A4_DEBUG_ASSERT(! owned_nmbr);
  A4_DEBUG_ASSERT(! global_nmbr);
  owned_nmbr = apf::numberOwnedNodes(mesh, "owned", shape);
  global_nmbr = apf::makeGlobal(owned_nmbr, false);
  apf::getNodes(global_nmbr, owned);
  auto num_owned = owned.getSize();
  Teuchos::Array<GO> indices(num_owned);
  for (size_t n = 0; n < num_owned; ++n)
    indices[n] = apf::getNumber(global_nmbr, owned[n]);
  node_map = Tpetra::createNonContigMap<LO, GO>(indices, comm);
  indices.resize(num_eqs * num_owned);
  for (size_t n = 0; n < num_owned; ++n)
```

```
  {
    GO gid = apf::getNumber(global_nmbr, owned[n]);
    for (int eq = 0; eq < num_eqs; ++eq)
      indices[get_dof(n, eq, num_eqs)] = get_gdof(gid, eq, num_eqs);
  }
  owned_map = Tpetra::createNonContigMap<LO, GO>(indices, comm);
  apf::synchronize(global_nmbr);
}

void Disc::compute_ghost_map()
{
  A4_DEBUG_ASSERT(! ghost_nmbr);
  ghost_nmbr = apf::numberOverlapNodes(mesh, "ghost", shape);
  apf::DynamicArray<apf::Node> ghost;
  apf::getNodes(ghost_nmbr, ghost);
  auto num_ghost = ghost.getSize();
  Teuchos::Array<GO> indices(num_eqs * num_ghost);
  for (size_t n = 0; n < num_ghost; ++n)
  {
    GO gid = apf::getNumber(global_nmbr, ghost[n]);
    for (int eq = 0; eq < num_eqs; ++eq)
      indices[get_dof(n, eq, num_eqs)] = get_gdof(gid, eq, num_eqs);
  }
  ghost_map = Tpetra::createNonContigMap<LO, GO>(indices, comm);
}

void Disc::compute_graphs()
{
  owned_graph = rcp(new Graph(owned_map, 300));
  ghost_graph = rcp(new Graph(ghost_map, 300));
  apf::MeshEntity* elem;
  auto elems = mesh->begin(num_dims);
  while ((elem = mesh->iterate(elems)))
  {
    apf::NewArray<long> gids;
```

```
    int nnodes = apf::getElementNumbers(global_nmbr, elem, gids);
    for (int i = 0; i < nnodes; ++i)
    {
      for (int j = 0; j < num_eqs; ++j)
      {
        GO row = get_gdof(gids[i], j, num_eqs);
        for (int k = 0; k < nnodes; ++k)
        {
          for (int l = 0; l < num_eqs; ++l)
          {
            GO col = get_gdof(gids[k], l, num_eqs);
            auto col_av = Teuchos::arrayView(&col, 1);
            ghost_graph->insertGlobalIndices(row, col_av);
          }
        }
      }
    }
  }
  mesh->end(elems);
  ghost_graph->fillComplete();
  auto exporter = rcp(new Export(ghost_map, owned_map));
  owned_graph->doExport(*ghost_graph, *exporter, Tpetra::INSERT);
  owned_graph->fillComplete();
  apf::destroyGlobalNumbering(global_nmbr);
}

void Disc::compute_coords()
{
  coords = rcp(new MultiVector(node_map, num_dims, false));
  apf::Vector3 point(0, 0, 0);
  for (size_t n = 0; n < owned.size(); ++n)
  {
    mesh->getPoint(owned[n].entity, owned[n].node, point);
    for (int d = 0; d < num_dims; ++d)
      coords->replaceLocalValue(n, d, point[d]);
```

```
  }
}

void Disc::compute_side_sets()
{
  int nss = sets->models[num_dims-1].size();
  for (int i = 0; i < nss; ++i)
    side_sets[get_side_set_name(i)].resize(0);
  apf::MeshIterator* it = mesh->begin(num_dims-1);
  apf::MeshEntity* side;
  while ((side = mesh->iterate(it)))
  {
    apf::ModelEntity* me = mesh->toModel(side);
    if (!sets->invMaps[num_dims-1].count(me))
      continue;
    apf::StkModel* fs = sets->invMaps[num_dims-1][me];
    std::string const& fsn = fs->stkName;
    apf::Up adjElems;
    mesh->getUp(side, adjElems);
    A4_DEBUG_ASSERT(adjElems.n == 1);
    side_sets[fsn].push_back(side);
  }
  mesh->end(it);
}

void Disc::compute_node_sets()
{
  auto nns = sets->models[0].size();
  for (size_t s = 0; s < nns; ++s)
    node_sets[get_node_set_name(s)].resize(0);
  for (size_t n = 0; n < owned.size(); ++n)
  {
    auto node = owned[n];
    auto ent = node.entity;
    std::set<apf::StkModel*> mset;
    apf::collectEntityModels(
        mesh, sets->invMaps[0], mesh->toModel(ent), mset)↩
          ;
    if (mset.empty()) continue;
    APF_ITERATE(std::set<apf::StkModel*>, mset, mit)
```

```
      {
         auto ns = *mit;
         auto nsn = ns->stkName;
         node_sets[nsn].push_back(node);
      }
   }
}

} // end namespace A4
```

A.10 A4_ElasticStiffness.hpp

```
#ifndef A4_ELASTIC_STIFFNESS_HPP
#define A4_ELASTIC_STIFFNESS_HPP

/// @file A4_Elastic_Stiffness.hpp

#include <apf.h>
#include <apfDynamicMatrix.h>

namespace A4 {

/// @brief Class to compute element stiffness matrix for linear elasticity.
class ElasticStiffness : public apf::Integrator
{

  public:

    /// @brief Construct the elastic stiffness integrator.
    /// @param m The relevant apf::Mesh structure.
    /// @param o The numerical integration order of accuracy.
    /// @param E Elastic modulus.
    /// @param nu Poisson's ratio.
    ElasticStiffness(
        apf::Mesh* m,
        int o,
        double E,
        double nu);

    /// @brief Set up data as each new element is encountered.
    /// @param me The incoming mesh element.
    void inElement(apf::MeshElement* me);

    /// @brief Work to be done at a single integration point.
    /// @param p The parametric integration point coordinate.
```

```
    /// @param w The numerical integration point weight.
    /// @param dv The differential volume (det J) at the ↩
        integration point.
    /// @details Calling
    /// - this->process(apf::Mesh* m) or
    /// - this->process(apf::MeshElement* e)
    /// will provide this method with the appropriate ↩
        input parameters.
    void atPoint(apf::Vector3 const& p, double w, double ↩
        dv);

    /// @brief Finalize data as we leave each element.
    void outElement();

    /// @brief The element stiffness matrix.
    apf::DynamicMatrix Ke;

  private:

    int num_dims;
    int num_elem_nodes;
    int num_elem_dofs;

    apf::DynamicMatrix D;
    apf::DynamicMatrix B;
    apf::DynamicMatrix DB;
    apf::DynamicMatrix BT;
    apf::DynamicMatrix K_tmp;

    apf::NewArray<apf::Vector3> dN;

    apf::Mesh* mesh;
    apf::FieldShape* shape;
    apf::MeshElement* mesh_element;
};

} // end namespace A4

#endif
```

A.11 A4_ElasticStiffness.cpp

```
#include <apfMesh.h>
#include <apfShape.h>

#include "A4_ElasticStiffness.hpp"

namespace A4 {

static void fill_elast_tensor(apf::DynamicMatrix& D, double E, double v)
{
  // ASSUMES PLANE STRESS
  D.setSize(3, 3);
  D.zero();
  D(0,0) =  1;
  D(0,1) =  v;
  D(0,2) =  0;
  D(1,0) =  v;
  D(1,1) =  1;
  D(1,2) =  0;
  D(2,0) =  0;
  D(2,1) =  0;
  D(2,2) =  (1-v)/2;

  D*=(E/(1-v*v));
}

ElasticStiffness::ElasticStiffness(apf::Mesh* m, int o, double E, double nu)
    : apf::Integrator(o),
      num_elem_dofs(0),
      mesh(m),
      shape(0),
      mesh_element(0)
{
  num_dims = m->getDimension();
  shape = m->getShape();
  fill_elast_tensor(D, E, nu);
}

```

```
void ElasticStiffness::inElement(apf::MeshElement* me)
{
  mesh_element = me;

  auto ent = apf::getMeshEntity(mesh_element);
  auto type = mesh->getType(ent);
  auto es = shape->getEntityShape(type);
  num_elem_nodes = es->countNodes();
  num_elem_dofs = num_dims * num_elem_nodes;

  B.setSize(3, num_elem_dofs);
  BT.setSize(num_elem_dofs, 3);
  DB.setSize(3, num_elem_dofs);
  Ke.setSize(num_elem_dofs, num_elem_dofs);
  K_tmp.setSize(num_elem_dofs, num_elem_dofs);

  B.zero();
  Ke.zero();
  K_tmp.zero();
}

void ElasticStiffness::atPoint(apf::Vector3 const& p, ↩
    double w, double dv)
{
  apf::getGradBF(shape, mesh_element, p, dN);
  for (int i = 0; i < num_elem_nodes; ++i)
  {
    B(0,2*i)       = dN[i][0];

    B(1,2*i+1)     = dN[i][1];

    B(2,2*i)       = dN[i][1];
    B(2,2*i+1)     = dN[i][0];
  }

  apf::transpose(B, BT);
  apf::multiply(D, B, DB);
  apf::multiply(BT, DB, K_tmp);

  K_tmp *= w * dv;
```

```
  Ke += K_tmp;

}

void ElasticStiffness::outElement()
{
  mesh_element = 0;
}

} // end namespace A4
```

A.12 A4_FESolver.hpp

```
#ifndef A4_FE_SOLVER_HPP
#define A4_FE_SOLVER_HPP

/// @file A4_FESolver.hpp

#include <Teuchos_ParameterList.hpp>
#include "A4_Disc.hpp"
#include "A4_LinAlg.hpp"
#include "A4_PostProc.hpp"

namespace A4 {

using Teuchos::ParameterList;

/// @cond
class Disc;
class ElasticStiffness;
/// @endcond

/// @brief A class to solve a linear elastic FEM problem.
/// @details With modified RHS data to account for ↩
    dislocation effects.
class FESolver
{

  public:

    /// @brief Construct the solver object.
    /// @param d The relevant discretization object.
    /// @param p The valid FEM solve parameters.
    /// @param order The integration order.
    /// @param load  The body load (x,y,z).
    FESolver(Disc* d, ParameterList const& p, int order, ↩
        double load[3]);

    /// @brief Destroy the FEM object.
    ~FESolver();

    /// @brief Assemble and solve the linear FEM problem.
```

```
    void solve();

    /// @brief Assign the contents of the solution to the field.
    /// @param f The field to write displacements to.
    void set_disp_to_field(apf::Field* f);

    /// @brief Assign the contents of the forcing vector to the field.
    /// @param f The field to write tractions to.
    void set_force_to_field(apf::Field* f);

    /// @brief Cacluate the Cauchy stress and assign to field.
    /// @param f The field to write Cauchy stress to.
    void set_stress_to_field(apf::Field* f);

    /// @brief Calculates the error in displacement
    ///        approximation with an L2 norm.
    double get_error();

  private:

    void assemble_LHS();
    void assemble_RHS();

    Disc* disc;
    ParameterList params;
    LinAlg la;
    int int_order;

    double g[3];

    ElasticStiffness* LHS;
};

} // end namespace A4

#endif
```

A.13 A4_FESolver.cpp

```
#include <apf.h>
#include <apfMesh2.h>
#include <Teuchos_ParameterList.hpp>

#include "A4_Disc.hpp"
#include "A4_FESolver.hpp"
#include "A4_LinSolve.hpp"
#include "A4_ElasticStiffness.hpp"
#include "A4_DBCs.hpp"
#include "A4_NBCs.hpp"
#include "A4_BodyLoads.hpp"
#include "A4_PostProc.hpp"

namespace A4 {

static ParameterList get_valid_params()
{
  ParameterList p;
  p.set<double>("E", 0.0);
  p.set<double>("nu", 0.0);
  p.sublist("dirichlet bcs");
  p.sublist("traction bcs");
  p.sublist("linear algebra");
  return p;
}

FESolver::FESolver(
  Disc* d,
  ParameterList const& p,
  int order,
  double load[3]) :
    disc(d),
    params(p),
    la(d)
{
  params.validateParameters(get_valid_params(), 0);
  int_order = order;
  g[0] = load[0];
  g[1] = load[1];
```

```
  g[2] = load[2];


}

FESolver::~FESolver()
{
}

void FESolver::assemble_LHS()
{

  // get the mesh and problem parameters
  auto mesh = disc->get_apf_mesh();
  double E = params.get<double>("E");
  double nu = params.get<double>("nu");

  // elemental information
  apf::DynamicMatrix Ke;
  apf::DynamicVector Ke_row;
  std::vector<LO> lids;

  // create the stiffness matrix integrator
  LHS = new ElasticStiffness(mesh, int_order, E, nu);

  // iterate over all elements in the mesh
  apf::MeshEntity* ent;
  auto it = mesh->begin(mesh->getDimension());
  while ((ent = mesh->iterate(it))) {

    // create a mesh element to pass to the integrator
    apf::MeshElement* me = apf::createMeshElement(mesh, ↩
       ent);

    // integrate over the current element
    LHS->process( me);

    // get elemental stiffness information
    disc->get_ghost_lids( ent, lids);
    Ke = LHS->Ke;
```

```
    int num_rows = Ke.getRows();

    // add elemental stiffness rows into the full system
    for (int i = 0; i < num_rows; ++i)
    {
      LO row = lids[i];
      LHS->Ke.getRow(i, Ke_row);
      auto cols = Teuchos::arrayView(&(lids[0]), num_rows);
      auto values = Teuchos::arrayView(&(Ke_row[0]), num_rows);
      la.ghost->K->sumIntoLocalValues(row, cols, values);
    }

    // destroy the mesh element to prevent memory leaks
    apf::destroyMeshElement(me);
  }
  mesh->end(it);

  // destroy the stiffness matrix integrator
  delete LHS;
}

void FESolver::assemble_RHS()
{
  apply_nbcs( disc, &la, params, int_order);
  apply_body_loads( disc, &la, int_order, g);
  return;
}

void FESolver::solve()
{
  assemble_LHS();
  assemble_RHS();
  la.gather_K();
  la.gather_F();
  apply_dbcs( disc, &la, params);
  la.owned->K->fillComplete();
  auto solve_params = params.sublist("linear_algebra");
  solve_linear_system( solve_params, &la, disc);
```

```
  return;
}

void FESolver::set_disp_to_field( apf::Field* f)
{
  set_to_field( f, la.owned->U, disc);
  return;
}

void FESolver::set_force_to_field( apf::Field* f)
{
  set_to_field( f, la.owned->F, disc);
  return;
}

void FESolver::set_stress_to_field( apf::Field* f)
{
  double E = params.get<double>("E");
  set_Cauchy_stress( E, f, la.owned->U, disc);
  return;
}

double FESolver::get_error()
{
  double E = params.get<double>("E");
  double nu = params.get<double>("nu");
  return get_L2_error( g, la.owned->U, disc, E, nu);
}

}
```

A.14 A4_LinAlg.hpp

```
#ifndef A4_LIN_ALG_HPP
#define A4_LIN_ALG_HPP

/// @file A4_LinAlg.hpp

#include "A4_Defines.hpp"
#include "A4_Disc.hpp"

namespace A4 {

/// @brief A general linear algebra object container.
struct LinAlgData
{
  /// @brief The stiffness matrix.
  RCP<Matrix> K;
  /// @brief The solution vector.
  RCP<Vector> U;
  /// @brief The load vector.
  RCP<Vector> F;
};

/// @brief A container for parallel linear algebra data
class LinAlg
{

  public:

    /// @brief Construct the linear algebra object.
    /// @param d The relevant discretization object.
    LinAlg(Disc* disc);

    /// @brief Destroy the linear algebra object.
    /// @details This will destroy the owned and ghost data objects.
    ~LinAlg();

    /// @brief Transfer data from ghost->K to owned->K.
    /// @details This is called with the Tpetra::ADD directive.
```

```
    void gather_K();

    /// @brief Transfer data from owned->F to ghost->F.
    /// @details This is called with the Tpetra::ADD directive.
    void gather_F();

    /// @brief The owned linear algebra data.
    LinAlgData* owned;

    /// @brief The ghost linear algebra data.
    LinAlgData* ghost;

  private:

    RCP<Import> importer;
    RCP<Export> exporter;

};

/// @brief Create a linear algebra object.
/// @param d The relevant discretization object.
LinAlg* create_lin_alg(Disc* d);

/// @brief Destroy a linear algebra object.
/// @param l The linear algebra object to destroy.
void destroy_lin_alg(LinAlg* l);

} // end namespace A4

#endif
```

A.15 A4_LinAlg.cpp

```
#include "A4_Disc.hpp"
#include "A4_LinAlg.hpp"

namespace A4 {

LinAlg::LinAlg(Disc* d)
{
  owned = new LinAlgData;
  ghost = new LinAlgData;

  auto owned_map = d->get_owned_map();
  auto ghost_map = d->get_ghost_map();
  auto owned_graph = d->get_owned_graph();
  auto ghost_graph = d->get_ghost_graph();

  importer = rcp(new Import(owned_map, ghost_map));
  exporter = rcp(new Export(ghost_map, owned_map));

  owned->K = rcp(new Matrix(owned_graph));
  owned->U = rcp(new Vector(owned_map));
  owned->F = rcp(new Vector(owned_map));

  ghost->K = rcp(new Matrix(ghost_graph));
  ghost->U = rcp(new Vector(ghost_map));
  ghost->F = rcp(new Vector(ghost_map));
}

void LinAlg::gather_K()
{
  owned->K->doExport(*(ghost->K), *exporter, Tpetra::ADD);
}

void LinAlg::gather_F()
{
  owned->F->doExport(*(ghost->F), *exporter, Tpetra::ADD);
}

```

```
LinAlg::~LinAlg()
{
  delete ghost;
  delete owned;
}

LinAlg* create_lin_alg(Disc* d)
{
  return new LinAlg(d);
}

void destroy_lin_alg(LinAlg* l)
{
  delete l;
}

} // end namespace A4
```

A.16 A4_LinSolve.hpp

```
#ifndef A4_LIN_SOLVE_HPP
#define A4_LIN_SOLVE_HPP

/// @file A4_LinSolve.hpp

namespace Teuchos {
class ParameterList;
}

namespace A4 {

using Teuchos::ParameterList;

/// @brief Solve a linear system iteratively.
/// @param p A parameter list.
/// @param la The linear algebra data.
/// @param disc The relevant discretization object.
void solve_linear_system(
    ParameterList const& p,
    LinAlg* la,
    Disc* disc);

}

#endif
```

A.17 A4_LinSolve.cpp

```cpp
#include <BelosBlockGmresSolMgr.hpp>
#include <BelosLinearProblem.hpp>
#include <BelosTpetraAdapter.hpp>
#include <MueLu.hpp>
#include <MueLu_TpetraOperator.hpp>
#include <MueLu_CreateTpetraPreconditioner.hpp>

#include "A4_Control.hpp"
#include "A4_Disc.hpp"
#include "A4_LinAlg.hpp"
#include "A4_LinSolve.hpp"

namespace A4 {

using Teuchos::rcp;
using Teuchos::rcpFromRef;

typedef Tpetra::MultiVector<ST, LO, GO, KNode> MV;
typedef Tpetra::Operator<ST, LO, GO, KNode> OP;
typedef Tpetra::RowMatrix<ST, LO, GO, KNode> RM;
typedef Belos::LinearProblem<ST, MV, OP> LinearProblem;
typedef Belos::SolverManager<ST, MV, OP> Solver;
typedef Belos::BlockGmresSolMgr<ST, MV, OP> GmresSolver;
typedef Tpetra::Operator<ST, LO, GO, KNode> Prec;

static ParameterList get_valid_params() {
  ParameterList p;
  p.set<int>("krylov size", 0);
  p.set<int>("maximum iterations", 0);
  p.set<double>("tolerance", 0.0);
  p.set<int>("output frequency", 0);
  p.sublist("multigrid");
  p.set<std::string>("method", "");
  return p;
}

static ParameterList get_belos_params(ParameterList const& in) {
  ParameterList p;
```

```
  int max_iters = in.get<int>("maximum iterations");
  int krylov = in.get<int>("krylov size");
  double tol = in.get<double>("tolerance");
  p.set<int>("Block Size" , 1);
  p.set<int>("Num Blocks", krylov);
  p.set<int>("Maximum Iterations", max_iters);
  p.set<double>("Convergence Tolerance", tol);
  p.set<std::string>("Orthogonalization", "DGKS");
  if (in.isType<int>("output frequency")) {
    int f = in.get<int>("output frequency");
    p.set<int>("Verbosity", 33);
    p.set<int>("Output Style", 1);
    p.set<int>("Output Frequency", f);
  }
  return p;
}


void solve_linear_system(
    ParameterList const& in,
    LinAlg* la,
    Disc* disc)
{
  in.validateParameters(get_valid_params(), 0);
  auto belos_params = get_belos_params(in);
  ParameterList mg_params(in.sublist("multigrid"));
  auto K = la->owned->K;
  auto U = la->owned->U;
  auto F = la->owned->F;
  auto KK = (RCP<OP>)K;
  auto coords = disc->get_coords();
  auto P = MueLu::CreateTpetraPreconditioner(KK, mg_params, coords);
  auto problem = rcp(new LinearProblem(K, U, F));
  problem->setLeftPrec(P);
  problem->setProblem();
  auto solver = rcp(new GmresSolver(problem, rcpFromRef(belos_params)));
  auto dofs = U->getGlobalLength();
  print(" > linear system: num dofs %zu", dofs);
```

```
  auto t0 = time();
  solver->solve();
  auto t1 = time();
  auto iters = solver->getNumIters();
  print(" > linear system: solved in %d iterations", iters);
  if (iters >= in.get<int>("maximum iterations"))
    print(" >  WARNING: solve was incomplete!");
  print(" > linear system: solved in %f seconds", t1 - t0);
}

}
```

A.18 A4_NBCs.hpp

```
#ifndef A4_NBCS_HPP
#define A4_NBCS_HPP

/// @file A4_NBCs.hpp

#include "A4_Disc.hpp"
#include "apfDynamicVector.h"

namespace Teuchos {
class ParameterList;
}

namespace A4 {

using Teuchos::ParameterList;
class LinAlg;

/// @brief Precribe Neumann boundary conditions.
/// @param d  The discretization of node sets.
/// @param la The relavent linear algebra data.
/// @param p  The parameterlist defining bcs.
/// @param inte_order
///           The numerical integration order of accuracy.
void apply_nbcs(
  Disc* d,
  LinAlg* la,
  ParameterList const& p,
  int inte_order);

class elemTrac :
  public apf::Integrator
{
  public:

    /// @brief Construct the traction stiffness integrator.
    /// @param d       The discretization object.
    /// @param la      The linear algebra object.
```

```
    /// @param order Order of numerical integration accuracy.
    /// @param val   The component value of traction.
    /// @param eqn   The equation number to modify.
    elemTrac(
      Disc* d,
      int order,
      double val,
      int eqNumber);

    /// @brief Prepare each new element.
    /// @param elem The mesh element.
    void inElement( apf::MeshElement* element);

    /// @brief Operate at an integration point.
    /// @param p  The parametric coordinates of the integration point
    /// @param w  The integration point weight.
    /// @param dv The differential volume at the integration point.
    /// @deatils Calling
    ///  - this->process(apf::Mesh* m) or
    ///  - this->process(apf::MeshElement* elem)
    /// will provide the input parameters.
    void atPoint(
      apf::Vector3 const& p,
      double w,
      double dv);

    /// @brief Finalize data once done with the element.
    void outElement();

    /// @brief Get the elemental forcing vector.
    apf::DynamicVector get_fe()
      { return fe; }

  private:

    apf::DynamicVector fe;

```

```
    int num_elem_nodes;
    int num_elem_dofs;
    int num_dims;
    int eqn;
    double value;

    apf::Mesh* mesh;
    apf::FieldShape* shape;
    apf::MeshElement* elem;
};

}

#endif
```

A.19 A4_NBCs.cpp

```
#include "A4_NBCs.hpp"
#include "A4_Disc.hpp"
#include "A4_LinAlg.hpp"

#include <apfDynamicVector.h>
#include <apfShape.h>

#include <Teuchos_ParameterList.hpp>
#include <string>

namespace A4 {

using Teuchos::getValue;
using Teuchos::Array;

elemTrac::elemTrac(
      Disc* d,
      int order,
      double val,
      int eqNumber):
  apf::Integrator(order)
{
  value = val;
  eqn = eqNumber;
  mesh = d->get_apf_mesh();
  shape = mesh->getShape();
  num_dims = mesh->getDimension();
}

void elemTrac::inElement(apf::MeshElement* element)
{
  elem = element;
  auto ent = apf::getMeshEntity(element);
  auto type = mesh->getType(ent);
  auto es = shape->getEntityShape(type);
  num_elem_nodes = es->countNodes();
  num_elem_dofs = num_dims * num_elem_nodes;

  fe.setSize(num_elem_dofs);
```

```
  fe.zero();

  return;
}

void elemTrac::atPoint( apf::Vector3 const& p, double w, double dv)
{
  apf::NewArray<double> Ns;
  apf::getBF( shape, elem, p, Ns);
  apf::DynamicVector f_tmp (num_elem_dofs);
  f_tmp.zero();
  int ind = 0;

  for( int i =0; i < num_elem_nodes; ++i)
  {
    for( int j = 0; j < num_dims; ++j)
    {
      if( j == eqn)
      {
        ind = i*num_dims+j;
        f_tmp[ind] += Ns[i]*value*w*dv;
      }
    }
  }

  fe += f_tmp;
  return;
}

void elemTrac::outElement()
{
  elem = 0;
  return;
}


void apply_nbcs(
  Disc* d,
  LinAlg* la,
```

```
  ParameterList const& p,
  int inte_order)
{
  std::vector<LO> lids;
  auto nbcs = p.sublist( "traction_bcs");
  auto mesh = d->get_apf_mesh();

  for( auto it = nbcs.begin(); it != nbcs.end(); ++it)
  {
    auto entry = nbcs.entry(it);
    // Format is: {0, xmin, 0.0} (spatial_dof, side_set_name, value)
    auto info = getValue<Array<std::string> >( entry);
    int s_dof = std::stoi( info[0]);
    std::string sideSet = info[1];
    double value = std::stod( info[2]);

    auto tracInter = new elemTrac( d, inte_order, value, s_dof);
    auto faces = d->get_sides( sideSet);
    for( int i = 0; i< (int)faces.size(); ++i)
    {
      apf::MeshEntity* face = faces[i];
      apf::MeshElement* elem = apf::createMeshElement( mesh, face);

      tracInter->process(elem);

      d->get_ghost_lids(face, lids);
      auto fe = tracInter->get_fe();
      int numRows = fe.getSize();
      for(int i = 0; i < numRows; i++)
      {
        auto row = lids[i];
        auto t = fe[i];
        la->ghost->F->sumIntoLocalValue( row, t);
      }
      apf::destroyMeshElement( elem);
    }
    delete tracInter;
```

```
    }

    return;
}

}
```

A.20 A4_PostProc.hpp

```
#ifndef A4_POSTPROC_HPP
#define A4_POSTPROC_HPP

/// @file A4_PostProc.hpp

#include <apf.h>
#include <apfShape.h>
#include <apfMesh.h>
#include <apfNumbering.h>
#include <apfDynamicMatrix.h>
#include "A4_Defines.hpp"
#include "A4_Disc.hpp"
#include <iostream>

namespace A4{

/// @brief Sets the values in RCP<vector> v to field f.
/// @param f The field to write to (using apf::setVector()).
/// @param v The vector of values.
/// @param d The discretization for node information.
void set_to_field( apf::Field* f, RCP<Vector> v, Disc* d);

/// @brief Calculates the Cauchy stress tensor and sets tensor to field.
/// @brief E Young's Modulus of the material.
/// @param f The field to write to (using apf::setMatrix()).
/// @param U The displacement solution vector.
/// @param d The relevant discretization object.
void set_Cauchy_stress( double E, apf::Field* f, RCP<Vector> U, Disc* d);


/// @brief Computes the L_2 norm of the error in the
///        approximation of the solution U.
/// @param g The body load vector.
/// @param U The approximated solution vector.
```

```
/// @param d The discretization object.
/// @param E Young's modulus.
/// @param nu Poisson's ratio.
double get_L2_error(
      double* g,
      RCP<Vector> U,
      Disc* d,
      double E,
      double nu);

} // End namespace A4
#endif
```

A.21 A4_PostProc.cpp

```
#include "A4_PostProc.hpp"

#include <math.h>

namespace A4{

void set_to_field( apf::Field* f, RCP<Vector> v, Disc* d)
{
  auto u = v->get1dView();
  auto owned_nmbr = d->get_owned_numbering();
  apf::Vector3 value;
  int nsd = d->get_apf_mesh()->getDimension();

  apf::DynamicArray<apf::Node> nodes;
  apf::getNodes( owned_nmbr, nodes);
  for( size_t n = 0; n < nodes.size(); n++)
  {
    auto node = nodes[n];
    auto e = node.entity;
    auto nodeth = node.node;
    for (int i = 0; i < nsd; i++)
    {
      auto row = d->get_owned_lid( node, i);
      value[i] = u[row];
    }
    apf::setVector( f, e, nodeth, value);
  }
  apf::synchronize( f);
  return;
}

void get_elemental_solution(
    apf::DynamicVector& u_e,
    RCP<Vector> U,
    std::vector<LO> lids)
{
  auto u = U->get1dView();

  for( size_t i = 0; i < lids.size(); i++)
```

```
  {
    auto row = lids[i];
    u_e[i] = u[row];
  }
  return;
}

void nye_to_matrix_planeStress(
    apf::DynamicVector& v,
    apf::Matrix3x3& m)
{
  // Diagonal
  m[0][0] = v(0);
  m[1][1] = v(1);

  // Upper tri
  m[0][1] = v(2);

  // Lower tri
  m[1][0] = v(2);

  return;
}

void zero_3x3( apf::Matrix3x3 m)
{
  for( int i = 0; i < 3; i++)
  {
    for( int j = 0; j < 3; j++)
    {
      m[i][j] = 0;
    }
  }
  return;
}

void set_elemental_stress(
    apf::MeshElement* elem,
    Disc* d,
    apf::Field* f,
```

```
    RCP<Vector> U,
    double E)
{
  auto ent = apf::getMeshEntity( elem);
  auto mesh = d->get_apf_mesh();
  auto shape = mesh->getShape();
  auto inte_order = shape->getOrder();

  // Get element's node ID's
  std::vector<LO> lids;
  d->get_ghost_lids( ent, lids);

  apf::Vector3 para;
  apf::NewArray<apf::Vector3> dN;
  apf::DynamicMatrix B;
  B.setSize( 3, lids.size());
  B.zero();

  apf::DynamicVector u_e (lids.size());
  u_e.zero();
  get_elemental_solution( u_e, U, lids);
  apf::DynamicVector s_e (3);


  int num_ips = apf::countIntPoints( elem, inte_order);
  for( int i = 0; i < num_ips; i++)
  {
    s_e.zero();
    apf::getIntPoint( elem, inte_order, i, para);
    apf::getGradBF(shape, elem, para, dN);

    size_t num_nodes = shape->getEntityShape(mesh->
        getType(ent))->countNodes();
    for( size_t j = 0; j < num_nodes; ++j)
    {
      B(0,2*j)       = dN[j][0];

      B(1,2*j+1)     = dN[j][1];

      B(2,2*j)       = dN[j][1];
```

```
        B(2,2*j+1)    = dN[j][0];
      }
      apf::multiply( B, u_e, s_e);
      s_e *= E;
      apf::Matrix3x3 sigma;
      zero_3x3( sigma);
      nye_to_matrix_planeStress( s_e, sigma);
      apf::setMatrix( f, ent, i, sigma);
  }

  return;
}

void set_Cauchy_stress( double E, apf::Field* f, RCP<Vector> U, Disc* d)
{
  auto mesh = d->get_apf_mesh();

  // Iterate over each mesh region
  apf::MeshEntity* ent;
  apf::MeshIterator* ent_it = mesh->begin(mesh->getDimension());
  while ((ent = mesh->iterate(ent_it)))
  {
    auto elem = apf::createMeshElement( mesh, ent);
    set_elemental_stress( elem, d, f, U, E);
    apf::destroyMeshElement( elem);
  }
  mesh->end( ent_it);

  apf::synchronize( f);
  return;
}

void compare_analytical_solution(
    double* g,
    RCP<Vector> Error,
    Disc* d,
    double E,
    double nu)
```

```
{
  auto e = Error->get1dView();
  auto mesh = d->get_apf_mesh();
  int nsd = mesh->getDimension();

  auto o_n = d->get_owned_numbering();
  apf::DynamicArray<apf::Node> nodes;
  apf::getNodes( o_n, nodes);
  for( size_t n = 0; n < nodes.size(); n++)
  {
    auto node = nodes[n];
    auto ent = node.entity;
    auto nodeth = node.node;
    for( int i = 0; i < nsd; i++)
    {
      apf::Vector3 pos;
      mesh->getPoint( ent, nodeth, pos);
      auto row = d->get_owned_lid( node, i);
      double a = 0;
      if( i == 0)
      {
        a = (g[0]/E) * (1 * pos[0] - pos[0] * pos[0]);
      }
      else if( i == 1)
      {
        a = (g[1]*(-nu)/E) * (1 * pos[0] - pos[0] * pos↩
            [0]);
      }
      Error->sumIntoLocalValue( row, -a);
    }

  }
  std::cout << std::endl;
  return;
}

double get_L2_error(
    double* g,
    RCP<Vector> U,
    Disc* d,
```

```
      double E,
      double nu)
{
   // Create a vector for the error
   auto e = U;
   compare_analytical_solution( g, e, d, E, nu);
   double norm = e->norm2();
   double length = (double)e->getGlobalLength();
   double RMS = norm/(sqrt(length));
   return RMS;
}

} // End namespace A4
```

B Example Input and Control Files

B.1 Associations File

The associations files followed the following format:
$\{Set\_Type\}$ $\{Name\}$ $\{Number\_of\_Model\_Entites\}$
$\{Model\_Entity\_Order\}$ $\{Model\_Entity\_Tag\_Number\}$
$\{Model\_Entity\_Order\}$ $\{Model\_Entity\_Tag\_Number\}$
$\vdots$
$\{Model\_Entity\_Order\}$ $\{Model\_Entity\_Tag\_Number\}$
$\{Set\_Type\}$ $\{Name\}$ $\{Number\_of\_Model\_Entites\}$
$\vdots$

```
elem set box 1
2 0
node set xmin 1
1 1
node set ymin 1
1 0
node set xmax 1
1 2
node set ymax 1
1 3
side set xmin 1
1 1
side set ymin 1
1 0
side set xmax 1
1 2
side set ymax 1
1 3
```

B.2 Example .yaml File

```yaml
test example:
  E: 1000.0
  nu: 0.25
  dirichlet bcs:
    bc 1: [0, xmin, 0.0]
    bc 2: [1, ymin, 0.0]
  traction bcs:
   bc 1: [0, xmax, 100.0]
  linear algebra:
    method: GMRES
    maximum iterations: 200
    krylov size: 200
    tolerance: 1.0e-10
    multigrid:
      number of equations: 2
      verbosity: none
```

B.3 Example Input Script

```
./../build/src/a4      \
  quad_4.dmg           \
  quad_4.smb           \
  plane2D.txt          \
  fem.yaml             \
  quad_quad_1           \
  2                    \
  0.0  0.0  0.0
```

C Derivations

Nomenclature for all equations shown in this section matches that described in the main body of this report. See Section 2 for variable descriptions.

C.1 Strong to Weak Form Derivation

Given:

$$\sigma_{ij},_j - f_i = 0 \quad \text{and} \quad w_i \in H^1, w_i\Big|_{\Gamma_i^g} = 0 \quad \forall \quad i = 1(1)n_{sd}$$

$$\sigma_{ij},_j = c_{ijkm}\varepsilon_{km} = c_{ijkm}u_{k},_m$$

Solution:

$$\begin{aligned}
\int_\Omega w_i c_{ijkm} u_{k},_{mj}\, d\Omega - \int_\Omega w_i f_i d\Omega &= 0 \\
-c_{ijkm}(w_i u_{k},_{mj})\Big|_{\partial\Omega=\Gamma} + \int_\Omega c_{ijkm} w_{i},_j\, u_{k},_m\, d\Omega &= \int_\Omega w_i f_i d\Omega \\
-c_{ijkm}(w_i u_{k},_{mj})\Big|_{\Gamma_i^g} - c_{ijkm}(w_i u_{k},_{mj})\Big|_{\Gamma_i^h} + \int_\Omega c_{ijkm} w_{i},_j\, u_{k},_m\, d\Omega &= \int_\Omega w_i f_i d\Omega \\
0 - c_{ijkm}(w_i u_{k},_{mj})\Big|_{\Gamma_i^h} + \int_\Omega c_{ijkm} w_{i},_j\, u_{k},_m\, d\Omega &= \int_\Omega w_i f_i d\Omega \\
-w_i\Big|_{\Gamma_i^h} h_i + \int_\Omega c_{ijkm} w_{i},_j\, u_{k},_m\, d\Omega &= \int_\Omega w_i f_i d\Omega \\
\int_\Omega c_{ijkm} w_{i},_j\, u_{k},_m\, d\Omega &= \int_\Omega w_i f_i d\Omega + w_i\Big|_{\Gamma_i^h} h_i
\end{aligned}$$

C.2 Weak to Galerkin Form Derivation

Given:

$$a(X,Y) = \int_\Omega c_{ijkm} X_{i,j}\, Y_{k,m}\, d\Omega$$
$$b(X,Y) = \int_\Omega X_i Y_i d\Omega$$
$$u_i^h = v_i^h + g_i^h \quad g_i^h\Big|_{\Gamma_i^g} = g_i \quad v_i^h\Big|_{\Gamma_i^g} = 0$$

Solution:

$$\begin{aligned}
\int_\Omega c_{ijkm} w_{i,j}\, u_{k,m}\, d\Omega &= \int_\Omega w_i f_i d\Omega + w_i\Big|_{\Gamma_i^h} h_i \\
a(w,u) &= b(w,f) + w_i\Big|_{\Gamma_i^h} h_i \\
\text{let} \quad u &\approx u^h = v^h + g^h \\
w &\approx w^h \\
a(w^h,u^h) &= b(w^h,f) + w_i^h\Big|_{\Gamma_i^h} h_i \\
a(w^h,v^h) + a(w^h,g^h) &= b(w^h,f) + w_i^h\Big|_{\Gamma_i^h} h_i \\
a(w^h,v^h) &= b(w^h,f) + w_i^h\Big|_{\Gamma_i^h} h_i - a(w^h,g^h)
\end{aligned}$$

C.3 Galerkin to Matrix Form Derivation

Given:

$$a(w^h, v^h) = b(w^h, f) + w_i^h\Big|_{\Gamma_i^h} h_i - a(w^h, g^h)$$

$$\begin{aligned}
w_i^h &= \sum_{A=1}^{n} c_i^A N^A \\
v_i^h &= \sum_{A=1}^{n} d_i^A N^A \\
g_i^h &= \sum_{A=1}^{n} g_i^A N^A
\end{aligned}$$

Solution:

$$\begin{aligned}
a(c^A N^A, d^B N^B) &= b(c^A N^A, f) + c^A N^A\Big|_{\Gamma_i^h} h_i - a(c^A N^A, g^C N^C) \\
c^A a(N^A, N^B) d^B &= c^A b(N^A, f) + c^A N^A\Big|_{\Gamma_i^h} h_i - c^A a(N^A, N^C) g^C \\
a(N^A, N^B) d^B &= b(N^A, f) + N^A\Big|_{\Gamma_i^h} h_i - a(N^A, N^C) g^C
\end{aligned}$$

$$\begin{aligned}
\text{Let} \quad K_{AB} &= a(N^A, N^B) \\
F_A &= b(N^A, f) + N^A\Big|_{\Gamma_i^h} h_i - a(N^A, N^C) g^C
\end{aligned}$$

$$[K_{AB}]\{d_B\} = \{F_A\}$$