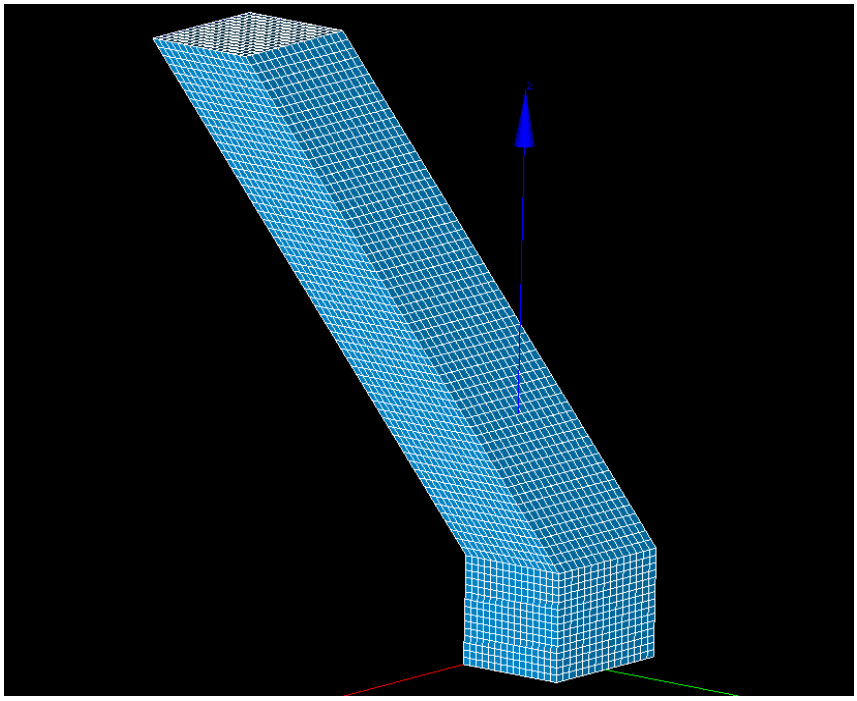
Additive Manufacturing

Mesh Generation:

Salome has been used for the mesh generation.



In salome, we have to set the meshing algorithms and hypothesis. For generating a structured hexahedral mesh as shown in the above figure, the following procedure is followed:

3D Algorithm: 3D Extrusion

2D Algorithm: Quadrangle Mapping

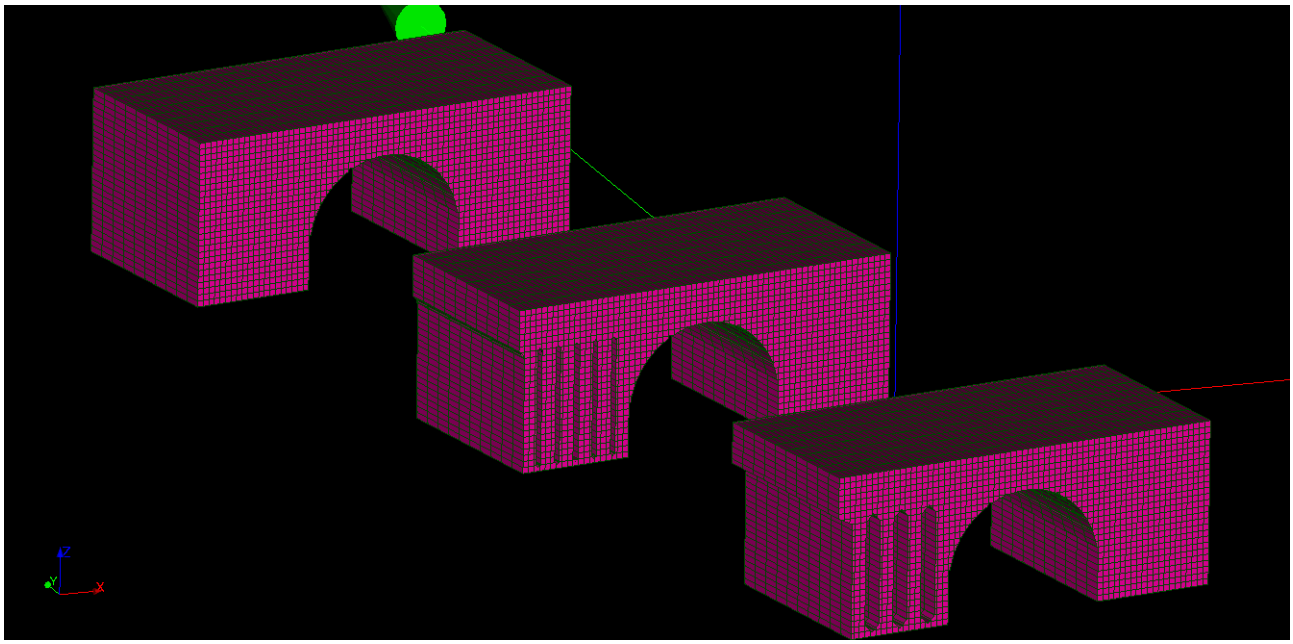
1D Algorithm: Wire Discretization with equidistant node distribution hypothesis. For detailed information please refer to the Salome manual on “basic meshing algorithms”.

To manage the required resolution of individual components, it is required to create the sub-meshing. For this particular case, we create two submeshes

- Quadrangle meshing of the base of the geometry
- Wire discretization for the slant edge with increased number of nodes and with hypothesis copy on the opposite edges of the geometry.

Unfortunately there is a problem with the 45 degree structure with the same procedure. Probably there is a small problem with the CAD model.

Meshing of the 3D bridge geometry



For this particular case of meshing the 3D geometry, we used body fititng 3D meshing algorithms.

Meshing Hypothesis:

In this case, wee choose the “Body fitting parameters” hypothesis. Then we can choose the normalized grid spacing in each axes direction. For additional options on the body fitting meshing, please refer to the basic meshing documentation on Salome.

Development of the finite element code:

Algorithmic implementation of the transient thermal problem

1. Read full-mesh from salome.
2. Assign cell iterators for individual layers.
3. Assign finite element index for individual layer.
4. Setup degree of freedom's, mass matrix, laplace matrix and RHS vector[created a linear system with matrix coefficients as constants such as (\rho*C_p)].
5. Assign boundary conditions (Dirichlet in-case of laser heating and Neumann during the cooling-phase)
6. Re-arrange the matrix system.
7. Solve the linear system for laser-heating process.
8. Repeat the steps 5-7 for the cooling process of individual-layers.
9. Plot the vtk plots to be viewed using paraview.

deal II documentation

On and off cells: This feature has been implemented using the FE_Nothing functionality of the deal.ii along with FE_Q(1) for the normal finite element cells.

- FE_Nothing- finite element with zero degree of freedom. So its basically a zero function over a finite element space. It indeed is useful in denoting the cell representation but the DOF's are not used for computation. For more on this feature, refer to the deal.II documentation.
- FE_Q(1)- a langrangian finite element defined on individual cell. In this case, the order of

the shape function in individual direction is linear.

Since there are two different finite elements involved in our system, it is assigned in the system using **hp::FECollection<dim>** fe_collection; object in our code.

Hence for each cell, we assign the finite element index accordingly

```
(*cell)->set_active_fe_index(0);
(*cell)->set_active_fe_index(1);
```

Layer-wise incrementation of the cells:

This has been achieved through layer-wise cell Iterators-

```
std::vector<cellsIterator>*  layerIterator  =  new  std::vector<cellsIterator>
[layers];
```

Then the fe_index in each individual layers is set according to the additive manufacturing process.

```
for (auto cell = layerIterator[currentLayer].begin();
            cell != layerIterator[currentLayer].end(); ++cell){
            (*cell)->set_active_fe_index(1); // 2 times de-referencing of
the cell because the iterator is one hop away from the data
            }
```

Setting-up the linear system(System Matrix, Solution, RHS Matrix)

(This has been implemented following the step-26 of deal.II library)

Setting up the DOF's using the class “Sparsity Pattern”

```
CompressedSparsityPattern
      compressed_sparsity_pattern(dof_handler.n_dofs());
sparsity_pattern.copy_from (compressed_sparsity_pattern);
```

The system matrix has been setup by using the functionality of deal.II library functions

```
MatrixCreator::create_mass_matrix(dof_handler,
      q_collection,
      mass_matrix,
(const Function<dim> *)& matrixCoefficient<dim>(rho*cP));
MatrixCreator::create_laplace_matrix(dof_handler,
      q_collection,
      laplace_matrix,
      (const Function<dim> *)& matrixCoefficient<dim>(kT));
```

Then proceeding towards assembling the system for the transient heat problem

```
\left( M + k_n \Theta A\right) U^n = M U^(n-1) - k_n (1-\Theta) A U^(n-1) +
k_n \left[ (1- \Theta)F^(n-1)+ \Theta F^(n)\right]
```

(written in latex context)

M*U^n−M*U^(n−1) + kn[(1−θ)*A*U^(n−1)+θ*A*U^n]=kn[(1−θ)*F^(n−1)+θ*F^n]

where “θ” suggests the time-stepping method used for the simulations.

If θ=0.5 represents crank-nicolson method.

And “kn” represents the individual time step dt

Implementing the boundary conditions

Assigning the heating boundary conditions based upon the boundary tags

```
VectorTools::interpolate_boundary_values (dof_handler,
      bottom_Bnd,
      ConstantFunction<dim>(80),
      boundary_values);
```

```
      VectorTools::interpolate_boundary_values (dof_handler,
                  currentLayer+layer_Bnd_offset,
                  ConstantFunction<dim>(1255),
                  boundary_values);
      // Modifying the system of equations
      MatrixTools::apply_boundary_values (boundary_values,
                  system_matrix,
                  solution,
                  system_rhs);
```

The complicated part is assigning the neumann boundary condition during cooling. Neumann boundary is directly implemented in the right hand side vector as shown:

```
      VectorTools::create_boundary_right_hand_side (dof_handler,
                                                    q_collection,
                                                    ZeroFunction<dim>(),
                                                    forcing_terms,
std::set<types::boundary_id>(layers, layers+2));
      system_rhs += forcing_terms;
```

As of now, zero function neumann boundary condition has been implemented, but we need to change it to heat-convection condition.

Solving the linear system

```
      Conjugate gradient method has been used for solving the system.
      SolverCG<> cg(solver_control);
      cg.solve(system_matrix, solution, system_rhs,
                  preconditioner);
```

```
Output results
Writing a vtk output for visulization of results on paraview
      DataOut<dim,hp::DoFHandler<dim> > data_out;
      data_out.attach_dof_handler(dof_handler);
      data_out.write_vtk(output);
```

Easy pitfalls to be kept in mind:

1. Can't assume a specific order in-which cells are accessed using the cell iterators. It has been assumed to be random.

2. The order of the DOF's is changed with adding additional layers as the manufacturing process proceeds.