SOLUTION of (13-1.p):

The volume-based formula for the dissipated power in the setting of this project is

$$P_{\text{diss}} = \int_{\Omega} |\frac{\partial v_1}{\partial x_2} - \frac{\partial v_2}{\partial x_1}|^2 \, \mathrm{d}\boldsymbol{x} \,, \tag{13.1.39}$$

into which we plug the finite element solution $\mathbf{v}_h \in \mathcal{S}_2^0(\mathcal{M})$.

The policy of the implementation is the same as for the member function `Eval()` in Sub-problem (13-1.h). We encapsulate the gradient of the local shape functions of $\mathcal{S}_2^0(\mathcal{M})$ in lambda functions and use the edge midpoint quadrature rule for the exact evaluation of the local integrals.

C++ code 13.1.40: Sub-problem (13-1.p): Volume formula for dissipated power ⇝ GitLab

```
double compDissPowVolume(const lf::assemble::DofHandler& dofh,
                         const Eigen::VectorXd& mu_vec) {
  // First fetch underlying mesh
  const lf::mesh::Mesh& mesh{*dofh.Mesh()};
  // Summation variable
  double p_diss{0.0};
  // Loop over all cells and add up contributions of local integrals to
     the
  // dissipated power.
  for (const lf::mesh::Entity* cell : mesh.Entities(0)) {
    LF_ASSERT_MSG(cell->RefEl() == lf::base::RefEl::kTria(),
                  "Only implemented for triangles");
    // Obtain coefficients for local shape functions
    // Local indices for LSFs fpr velocity components
    std::array<Eigen::Index, 6> vx_idx{0, 3, 6, 9, 11, 13};
    std::array<Eigen::Index, 6> vy_idx{1, 4, 7, 10, 12, 14};
    // !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
    // Workaround for a side-effect in LehrFEM++
    // Local shape functions associated with edges have to be swapped,
       if the
    // edge has "negative" orientation with respect to the triangle.
       This is
    // meant to offset such a swapping done by LehrFEM++'s
       UniformFEDofHandler.
    auto edge_orientations = cell->RelativeOrientations();
    LF_ASSERT_MSG(edge_orientations.size() == 3,
                  "Triangle should have 3 edges!?");
    for (int k = 0; k < 3; ++k) {
      if (edge_orientations[k] == lf::mesh::Orientation::negative) {
        // The rows and columns of the element matrix with numbers 9+2*k
           and
        // 9+2*k+1 have to be swapped.
        std::swap(vx_idx[3 + k], vy_idx[3 + k]);
      }
    }
    // !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
    std::span<const lf::assemble::gdof_idx_t> dof_idx{
        dofh.GlobalDofIndices(*cell)};
    LF_ASSERT_MSG(dof_idx.size() == 15, "Taylor-Hood FEM involves 15 LSFs!");

    // Copied from Eval(); Found no good way of moving this to a
       function
    // Area of the triangle
    double area = lf::geometry::Volume(*cell->Geometry());
```

```
    // Compute gradients of barycentric coordinate functions, see
    // [Lecture → Code 2.4.5.11] Get vertices of the triangle
    auto endpoints = lf::geometry::Corners(*(cell->Geometry()));
    Eigen::Matrix<double, 3, 3> X;  // temporary matrix
    X.block<3, 1>(0, 0) = Eigen::Vector3d::Ones();
    X.block<3, 2>(0, 1) = endpoints.transpose();
    // This matrix contains grad λ_i in its columns
    const auto G{X.inverse().block<2, 3>(1, 0)};
    // Gradients of local shape functions of quadratic Lagrangian finite
      element
    // space as lambda functions, see (13.1.24)
    std::array<std::function<Eigen::Vector2d(Eigen::Vector3d)>, 6> gradq{
        [&G](Eigen::Vector3d c) -> Eigen::Vector2d {
          return (4 * c[0] - 1) * G.col(0);
        },
        [&G](Eigen::Vector3d c) -> Eigen::Vector2d {
          return (4 * c[1] - 1) * G.col(1);
        },
        [&G](Eigen::Vector3d c) -> Eigen::Vector2d {
          return (4 * c[2] - 1) * G.col(2);
        },
        [&G](Eigen::Vector3d c) -> Eigen::Vector2d {
          return 4 * (c[0] * G.col(1) + c[1] * G.col(0));
        },
        [&G](Eigen::Vector3d c) -> Eigen::Vector2d {
          return 4 * (c[1] * G.col(2) + c[2] * G.col(1));
        },
        [&G](Eigen::Vector3d c) -> Eigen::Vector2d {
          return 4 * (c[2] * G.col(0) + c[0] * G.col(2));
        }};
    // Barycentric coordinates of the midpoints of the edges for
    // use with the 3-point edge midpoint quadrature rule (13.1.25)
    // Note that we only integrate a quadratic polynomial so that the
      edge
    // midpoint quadrature rule evaluates the integral exactly.
    const std::array<Eigen::Vector3d, 3> mp = {Eigen::Vector3d({0.5, 0.5, 0}),
                                               Eigen::Vector3d({0, 0.5, 0.5}),
                                               Eigen::Vector3d({0.5, 0, 0.5})};
    // We have to integrate |∂v_2/∂x_1 − ∂v_1/∂x_2|^2.
    // Loop over quadrature points
    double s = 0.0;
    for (int qp_idx = 0; qp_idx < 3; ++qp_idx) {
      // Compute grad v_1 and grad v_2 quadrature point
      Eigen::Vector2d grad_vx{0.0, 0.0};
      Eigen::Vector2d grad_vy{0.0, 0.0};
      for (int i = 0; i < 6; ++i) {
        grad_vx += mu_vec[dof_idx[vx_idx[i]]] * gradq[i](mp[qp_idx]);
        grad_vy += mu_vec[dof_idx[vy_idx[i]]] * gradq[i](mp[qp_idx]);
      }
      double curl_qp = grad_vy[0] - grad_vx[1];
      s += curl_qp * curl_qp;
    }
    p_diss += (area / 3.0 * s);
  }
  return p_diss;
}
```