C++ code 3.17.7: Implementation of `volumeResiduals()` ↬ GitLab

```cpp
lf::mesh::utils::CodimMeshDataSet<double> volumeResiduals(
    const dataDiscreteBVP &disc_bvp, const Eigen::VectorXd & /*u_vec*/) {
  // Get pointer to underlying mesh
  std::shared_ptr<const lf::mesh::Mesh> mesh_p =
      disc_bvp.pwlinfespace_p_->Mesh();
  // Mesh data set for returning the volume residuals
  lf::mesh::utils::CodimMeshDataSet<double> vol_res(mesh_p, 0, 0.0);

  // Edge midpoint quadrature rule for triangles, which is exact for
     quadratic
  // polynomials and allows the exact computation of the volume residual
  // contributions for piecewise linear source functions.
  const lf::quad::QuadRule qr{lf::quad::make_TriaQR_EdgeMidpointRule()};
  const Eigen::VectorXd qw{qr.Weights()};  // quadrature weigth vector

  // Run over all cells of the mesh
  for (const lf::mesh::Entity *cell : mesh_p->Entities(0)) {
    LF_ASSERT_MSG(cell->RefEl() == lf::base::RefEl::kTria(),
                  "Implemented for triangles only");
    // Obtain information about the shape of the cell
    const lf::geometry::Geometry &geo{*(cell->Geometry())};
    // Ddetermine size of triangle (length of longest edge)
    const Eigen::MatrixXd corners{lf::geometry::Corners(geo)};
    const double h0 = (corners.col(1) - corners.col(0)).norm();
    const double h1 = (corners.col(2) - corners.col(1)).norm();
    const double h2 = (corners.col(0) - corners.col(2)).norm();
    const double h_K = std::max({h0, h1, h2});

    // I: Locally integrate the square of the source function
    // Gramian determinants
    const Eigen::VectorXd gram_dets{geo.IntegrationElement(qr.Points())};
    // Fetch values of source function in quadrature nodes.  Note that
       local
    // reference coordinates have to be passed to the evaluation
       operator of the
    // MeshFunction.
    const std::vector<double> f_vals{disc_bvp.mf_f_(*cell, qr.Points())};
    // Evaluation of quadrature formula
    double f_sq_int = 0.0;
    for (lf::base::size_type l = 0; l < qr.NumPoints(); ++l) {
      f_sq_int += qw[l] * f_vals[l] * f_vals[l] * gram_dets[l];
    }
    // II: Fetch prefactor depending on diffusion coefficient
    // We need a "dummy point" in the reference element in order to call
       the
    // evaluation operator for the MeshFunction.
    const Eigen::MatrixXd dummy = Eigen::Vector2d(1.0 / 3.0, 1.0 / 3.0);
    const double alpha_K = disc_bvp.mf_alpha_(*cell, dummy)[0];
    LF_ASSERT_MSG(alpha_K > 0,
                  "Diffusion coefficients must be strictly positive!");
    const double mu_K = h_K * h_K / alpha_K;
    // Store cell contribution
    vol_res(*cell) = f_sq_int * mu_K;
  }
  return vol_res;
}
```