**C++11 code 2.12.3: Sub-problem (2-12.c): function `testQuadOrderQuad()` ↪ GitLab**

```cpp
bool testQuadOrderQuad(const lf::quad::QuadRule &quad_rule,
                       unsigned int order) {
  bool order_isExact = true;  // return variable

  double my_epsilon = 1e-12;
  // Retrieve the passed quadrature rule's reference element
  // NOLINTBEGIN(clang-analyzer-deadcode.DeadStores)
  const lf::base::RefEl ref_element = quad_rule.RefEl();
  // NOLINTEND(clang-analyzer-deadcode.DeadStores)
  // Check that the passed reference element is triangular
  assert(ref_element == lf::base::RefElType::kQuad);
  // A quadrature rule consists of quadrature nodes and weights defined so that
  // the weighted sum of the value of a function at these points approximates
  // the integral of that function.
  const Eigen::VectorXd weights = quad_rule.Weights();
  const Eigen::MatrixXd points = quad_rule.Points();  // (x,y) points
  const Eigen::VectorXd x_coords = points.row(0);
  const Eigen::VectorXd y_coords = points.row(1);
  // A quadrature rule over a two dimensional domain is of order k if it can
  // integrate exactly all bivariate polynomials of order k-1. The collection of
  // such polynomials is spanned by the set of homogeneous polynomials of order
  // strictly less than k, i.e. by the polynomials of the form
  /* p_IJ(x,y) = (1-x)^I(y^J), I,J < k: */
  auto eval_p_IJ = [&x_coords, &y_coords](int I, int J) -> Eigen::VectorXd {
    return x_coords.array().pow(I) * y_coords.array().pow(J);
  }; /* evaluates p_IJ at all points (x,y) as defined by quad_rule */

  /* Compare the analytical value and the quadrature sum for all p_IJ, I+J < k
   */
  double exact_integral;  // analytical value of the integral
  double quad_rule_sum;   // weighted sum used for approximating the integral
  for (int I = 0; I < order; I++) {
    for (int J = 0; J < order; J++) {
      exact_integral = 1.0 / ((I + 1.0) * (J + 1.0));
      quad_rule_sum = eval_p_IJ(I, J).dot(weights);

      // Check if the difference bewteen the results is within tolerance
      order_isExact = fabs(exact_integral - quad_rule_sum) <=
                      fabs(exact_integral) * my_epsilon;
      if (!order_isExact) {
        return order_isExact;
      }
    }
  }
  return order_isExact;
}
```