C++ code 9.12.8: Implementation of `(solveRKSobEv)` ➙ GitLab

```cpp
template <typename MESHFUNCTION_BETA, typename MESHFUNCTION_ALPHA>
Eigen::VectorXd solveRKSobEvl(
    std::shared_ptr<const lf::fe::ScalarFESpace<double>> fe_space_p,
    const MESHFUNCTION_BETA &beta, const MESHFUNCTION_ALPHA &alpha,
    const Eigen::VectorXd &mu0, double T, const Eigen::MatrixXd &RK_Mat,
    const Eigen::VectorXd &b, unsigned int M) {
  LF_ASSERT_MSG(mu0.size() == (fe_space_p->LocGlobMap()).NumDofs(),
                "Wrong length of coefficient vector");
  // Build Galerkin matrices taking into account homogeneous Dirichlet boundary
  // conditions
  Eigen::SparseMatrix<double> B =
      getFEMatrixDirichlet<double, MESHFUNCTION_BETA>(fe_space_p, beta);
  Eigen::SparseMatrix<double> A =
      getFEMatrixDirichlet<double, MESHFUNCTION_ALPHA>(fe_space_p, alpha);
  Eigen::VectorXd muj(mu0);  // current state vector
  // LU-decompose B outside timestepping loop, which is important for efficient
  // implementation
  Eigen::SparseLU<Eigen::SparseMatrix<double>> solver;
  solver.compute(B);
  LF_VERIFY_MSG(solver.info() == Eigen::Success, "LU decomposition failed");

  int s = RK_Mat.cols();  // Number of stages
  LF_ASSERT_MSG(s == RK_Mat.rows(), "Butcher matrix must be square!");
  LF_ASSERT_MSG(s == b.size(), "s weights required!");
  // timestep size
  double tau = T / M;
  // Main timestepping loop
  Eigen::VectorXd mu_next(mu0.size());           // next state vector
  Eigen::VectorXd tmp(mu0.size());               // Summation vector
  std::vector<Eigen::VectorXd> incs(s, mu_next); // RK increments
  // The r.h.s.  vector field for MOL ODE in standard form
  // Invoking solver.solve() amounts to the application of the
  // inverse of the matrix B.
  auto Vf = [&solver, &A](const Eigen::VectorXd &y) -> Eigen::VectorXd {
    const Eigen::VectorXd fy = -solver.solve(A * y);
    LF_VERIFY_MSG(solver.info() == Eigen::Success, "Solving LSE failed");
    return fy;
  };
  for (int k = 0; k < M; ++k) {
    // First increment vector
    incs[0] = Vf(muj);
    // Compute increments successively, which is possible for an explicit RK-SSM
    // Simultaneous assemble next state vector
    mu_next = muj + (tau * b[0]) * incs[0];
    for (int i = 1; i < s; ++i) {
      // Weighted sum of already computed increments
      tmp.setZero();
      for (int j = 0; j < i; ++j) {
        tmp += RK_Mat(i, j) * incs[j];
      }
      incs[i] = Vf(muj + tau * tmp);
      // Update state by weighed sum of increments
      mu_next += (tau * b[i]) * incs[i];
    }
```