C++ code 9.10.29: Code for `leapfrogMixedWave()` → GitLab

```
template <typename RHOFUNCTION, typename FFUNCTION,
          typename RECORDER = std::function<void(double, double)>>
std::pair<Eigen::VectorXd, Eigen::VectorXd> leapfrogMixedWave(
    const std::shared_ptr<lf::uscalfe::FeSpaceLagrangeO1<double>> &fe_space_V,
    const lf::assemble::UniformFEDofHandler &dofh_Q, RHOFUNCTION &&rho,
    FFUNCTION &&f, double T, unsigned int nb_timesteps,
    RECORDER &&rec = [](double, double) {}) {
  // Size of timestep
  double stepsize = T / nb_timesteps;
  // Index mapper for linear Lagrangian FE space
  const lf::assemble::DofHandler &dofh_V{fe_space_V->LocGlobMap()};
  // Dimension of finite element space
  const lf::uscalfe::size_type N_dofs_Q(dofh_Q.NumDofs());
  const lf::uscalfe::size_type N_dofs_V(dofh_V.NumDofs());

  // Zero INITIAL CONDITIONS
  Eigen::VectorXd mu_init = Eigen::VectorXd::Zero(N_dofs_V);
  Eigen::VectorXd kappa_init = Eigen::VectorXd::Zero(N_dofs_Q);

// PRECOMPUTATIONS: essential for efficiency
  std::cout << "Precomputing Galerkin Matrices :" << std::endl;
  // Galerkin matrix M̃_V
  Eigen::SparseMatrix<double> M_V = computeMV(fe_space_V, rho);
  std::cout << "Creating SparseLU solver for M_V: ";
  Eigen::SparseLU<Eigen::SparseMatrix<double>> solver_MV;
  solver_MV.compute(M_V);
  LF_VERIFY_MSG(solver_MV.info() == Eigen::Success, "LU decomposition failed");
  std::cout << " Done" << std::endl;
  // Galerkin matrix M̃_Q
  Eigen::SparseMatrix<double> M_Q = computeMQ(dofh_Q);
  std::cout << "Creating SparseLU solver for M_Q: ";
  Eigen::SparseLU<Eigen::SparseMatrix<double>> solver_MQ;
  solver_MQ.compute(M_Q);
  LF_VERIFY_MSG(solver_MQ.info() == Eigen::Success, "LU decomposition failed");
  std::cout << "Done" << std::endl;
  // Galerkin matrix B̃
  Eigen::SparseMatrix<double> B = computeB(dofh_V, dofh_Q);

  // EVOLUTION
  // Vectors for returning result
  Eigen::VectorXd mu;
  Eigen::VectorXd kappa;
  std::cout << "Evolution using the leapfrog timestepping scheme." << std::endl;
  Eigen::VectorXd mu_cur = mu_init;        // μ⃗^(j)
  Eigen::VectorXd kappa_cur = kappa_init;  // κ⃗^(j+1/2)
  Eigen::VectorXd mu_next;                 // μ⃗^(j+1)
  Eigen::VectorXd kappa_next;              // κ⃗^(j+3/2)
  Eigen::VectorXd kappa_avg;
  for (int j = 0; j < nb_timesteps; j++) {
    // Right hand side vector φ⃗(τ(j+1/2))
    const Eigen::VectorXd rhs{computeRHS(fe_space_V, f, stepsize * (j + 0.5))};
    // Update of μ⃗
    mu_next = solver_MV.solve(stepsize * (rhs + B.transpose() * kappa_cur) +
                              M_V * mu_cur);

    // Update of κ⃗
```