C++ code 9.4.17: Sub-problem (9-4.c): function `waveLeapfrogABC()` ⇝ GitLab

```
Eigen::MatrixXd waveLeapfrogABC(double c, double T, unsigned int N,
                                unsigned int m) {
  // N is also the number of cells of the mesh
  double h = 1.0 / N;
  // Obtain Galerkin matrices for truncated finite element space
  // Note that the functions get*_full return the matrices for the full finite
  // element space including the tent function located at x=1. Removing the last
  // row and column of that matrix amounts to dropping that basis function.
  // However, the efficiency of this block() operation in the case of sparse
  // matrices is in doubt, in particular, since the result is assigned to
  // another sparse matrix, which foils Eigen's expression template
  // optimization. The use of "auto" would be highly advisable here!
  Eigen::SparseMatrix<double> A = getA_full(N, c, h).block(0, 0, N, N);
  Eigen::SparseMatrix<double> B = getB_full(N, c).block(0, 0, N, N);
  Eigen::SparseMatrix<double> M = getM_full(N, h).block(0, 0, N, N);
  // Matrix for returning solution
  Eigen::MatrixXd R(m + 1, N + 1);
  Eigen::VectorXd mu = Eigen::VectorXd::Zero(N);    // = mu^(0)
  Eigen::VectorXd nu = Eigen::VectorXd::Zero(N);    // = nu^(-1/2)
  Eigen::VectorXd phi = Eigen::VectorXd::Zero(N);   // = phi(t_0)
  // Universally zero initial conditions make it possible to skip
  // the special initial step usually required for leapfrog.
  double tau = T / m;  // Timestep size
  // The diagonal matrix to be "inverted" in each timestep
  Eigen::SparseLU<Eigen::SparseMatrix<double>> solver(M + 0.5 * tau * B);
  for (int j = 0; j < m; ++j) {
    R.row(j).head(N) = mu.transpose();
    phi(N - 1) = c * c / h * g(j * tau);
    // Maybe, this can be done more efficiently by extracting the diagonal,
    // converting it into an Eigen::Array object and then perform componentwise
    // division. However, a really smart sparse elimination solver should be
    // able to detect a diagonal coefficient matrices and optimize the
    // elimination accordingly.
    nu = solver.solve(-tau * A * mu + (M - 0.5 * tau * B) * nu + tau * phi);
    mu = mu + tau * nu;
  }
  R.row(m).head(N) = mu.transpose();
  // The value at x=1 has to be incorporated into the output
  for (int i = 0; i < m + 1; ++i) {
    R(i, N) = g(i * tau);
  }
  return R;
}
```