C++ code 9.13.27: Implementation of `solve1DWavePML()`

```
template <typename RECORDER = std::function<void(const Eigen::VectorXd &)>>
Eigen::VectorXd solve1DWavePML(
    const Eigen::VectorXd &zeta_0, const Eigen::VectorXd &gamma,
    const Eigen::VectorXd &sigma, unsigned int M, double T,
    RECORDER &&rec = [](Eigen::VectorXd & /*zeta*/) -> void {}) {
  // Grid resolution parameter N = number of grid nodes - 1
  const unsigned int N = gamma.size() - 1;
  assert(N == sigma.size() - 1);
  assert(2 * N + 1 == zeta_0.size());
  // Vector of basis expansion coefficients, contains
  // ζ⃗^(k)
  Eigen::VectorXd zeta{zeta_0};
  // Obtain initial velocities at grid nodes by linear interpolation
  Eigen::VectorXd v0 =
      (Eigen::VectorXd(N + 1) << 0.0,
       0.5 * (zeta.segment(N + 1, N - 1) + zeta.segment(N + 2, N - 1)), 0.0)
          .finished();
  // Discretization parameters
  const double L = L_default;          // default width of PML layer
  const double h = (2.0 + 2 * L) / N;  // meshwidth
  const double tau = T / M;            // size of timestep
  // I. Initialize sparse matrices A, R ∈ ℝ^{2N+1,2N+1}.
  Eigen::SparseMatrix<double> A(2 * N + 1, 2 * N + 1);
  Eigen::SparseMatrix<double> R(2 * N + 1, 2 * N + 1);
  // We know that the matrix A has at most 3 non-zero entries per
     row/column
  A.reserve(Eigen::VectorXi::Constant(2 * N + 1, 3));
  R.reserve(Eigen::VectorXi::Constant(2 * N + 1, 3));
  // First initialize diagonal
  A.insert(0, 0) = h / (2 * tau) + 0.25 * h * sigma[0];
  R.insert(0, 0) = -h / (2 * tau) + 0.25 * h * sigma[0];
  for (unsigned int i = 1; i < N; ++i) {
    A.insert(i, i) = h / tau + 0.5 * h * sigma[i];
    R.insert(i, i) = -h / tau + 0.5 * h * sigma[i];
  }
  A.insert(N, N) = h / (2 * tau) + 0.25 * h * sigma[N];
  R.insert(N, N) = -h / (2 * tau) + 0.25 * h * sigma[N];
  for (unsigned int i = 0; i < N; ++i) {
    A.insert(i + N + 1, i + N + 1) =
        h / tau + 0.25 * h * (sigma[i] + sigma[i + 1]);
    R.insert(i + N + 1, i + N + 1) =
        -h / tau + 0.25 * h * (sigma[i] + sigma[i + 1]);
    A.insert(i, i + N + 1) = -0.5;
    A.insert(i + 1, i + N + 1) = 0.5;
    R.insert(i, i + N + 1) = -0.5;
    R.insert(i + 1, i + N + 1) = 0.5;
  }
  for (unsigned int j = 0; j < N; ++j) {
    A.insert(j + N + 1, j) = 0.25 * (gamma[j] + gamma[j + 1]);
    A.insert(j + N + 1, j + 1) = -0.25 * (gamma[j] + gamma[j + 1]);
    R.insert(j + N + 1, j) = 0.25 * (gamma[j] + gamma[j + 1]);
    R.insert(j + N + 1, j + 1) = -0.25 * (gamma[j] + gamma[j + 1]);
  }
  // For the sake efficiency Precompute LU factorization of A
  Eigen::SparseLU<Eigen::SparseMatrix<double>> solver;
  solver.compute(A);
```