C++11 code 2.2.12: Solution code for Sub-problem (2-2.e) ⇢ GitLab

```cpp
std::vector<Eigen::Triplet<double>> transformCOOmatrix(
    const std::vector<Eigen::Triplet<double>> &A) {
  std::vector<Eigen::Triplet<double>> A_t{};  // return value

  // First step:  find the size of the matrix by searching the maximal
  // indices.  Depends on the assumption that no zero rows/columns
     occur.
  int rows_max_idx = 0, cols_max_idx = 0;
  for (const Eigen::Triplet<double> &triplet : A) {
    rows_max_idx =
        (triplet.row() > rows_max_idx) ? triplet.row() : rows_max_idx;
    cols_max_idx =
        (triplet.col() > cols_max_idx) ? triplet.col() : cols_max_idx;
  }
  int n_cols = cols_max_idx + 1;

  // Make sure we deal with a square matrix
  assert(rows_max_idx + 1 == n_cols);
  // The matrix size must have even parity
  assert(n_cols % 2 == 0);

  int N = n_cols;  // Size of (square) matrix
  int M = N / 2;   // Half the size
  // clang-format off
  // Distribute entries of "old" matrix to new matrix
  for (const Eigen::Triplet<double> &it : A) {
    // row and column indices of current A triplet
    const int I = it.row() + 1; // k in (2.2.8)-(2.2.11)
    const int J = it.col() + 1; // ℓ in (2.2.8)-(2.2.11)

    // Distinguish different parities of indices
    if (I % 2 == 0 & J % 2 == 0) {
      // even, even, (2.2.8)
      A_t.emplace_back(I / 2 - 1, J / 2 - 1, it.value());
      A_t.emplace_back(I / 2 + M - 1, J / 2 + M - 1, it.value());
      A_t.emplace_back(I / 2 + M - 1, J / 2 - 1, -it.value());
      A_t.emplace_back(I / 2 - 1, J / 2 + M - 1, -it.value());
    } else if (I % 2 != 0 & J % 2 != 0) {
      // odd, odd, see (2.2.9)
      A_t.emplace_back((I + 1) / 2 - 1, (J + 1) / 2 - 1, it.value());
      A_t.emplace_back((I + 1) / 2 + M - 1, (J + 1) / 2 + M - 1, it.value());
      A_t.emplace_back((I + 1) / 2 - 1, (J + 1) / 2 + M - 1, it.value());
      A_t.emplace_back((I + 1) / 2 + M - 1, (J + 1) / 2 - 1, it.value());
    } else if (I % 2 == 0 & J % 2 != 0) {
      // even, odd, see (2.2.10)
      A_t.emplace_back(I / 2 - 1, (J + 1) / 2 - 1, it.value());
      A_t.emplace_back(I / 2 - 1, (J + 1) / 2 + M - 1, it.value());
      A_t.emplace_back(I / 2 + M - 1, (J + 1) / 2 + M - 1, -it.value());
      A_t.emplace_back(I / 2 + M - 1, (J + 1) / 2 - 1, -it.value());
    } else if (I % 2 != 0 & J % 2 == 0) {
      // odd, even, see (2.2.11)
      A_t.emplace_back((I + 1) / 2 - 1, J / 2 - 1, it.value());
      A_t.emplace_back((I + 1) / 2 + M - 1, J / 2 + M - 1, -it.value());
      A_t.emplace_back((I + 1) / 2 + M - 1, J / 2 - 1, it.value());
      A_t.emplace_back((I + 1) / 2 - 1, J / 2 + M - 1, -it.value());
    } else {
      assert(false);
```