

C++ code 2.19.11: Computing $\mathbf{M} \otimes \mathbf{M}$ for a sparse matrix $\mathbf{M}$

```

1 Eigen::SparseMatrix<double> sparseKron(const Eigen::SparseMatrix<double> &M) {
2     int n = M.rows();
3     assert(n == M.cols() && "Matrix M must be square.");
4
5     // JM is an array of size n+1, with
6     // JM[j] = number of non-zero entries in all columns 0,..., (j-1) of M.
7     const int *JM = M.outerIndexPtr();
8
9
10    // Define nnzColwiseM(j) = nnz( M.col(j) )
11    Eigen::VectorXi nnzColwiseM(n);
12    for (int j = 0; j < n; j++) {
13        nnzColwiseM(j) = JM[j + 1] - JM[j];
14    }
15
16    // B will be the Kronecker product of M with itself.
17    Eigen::SparseMatrix<double> B(n * n, n * n);
18
19    // TODO: (2-19.i) Fill in the entries of B.
20    // Hint: You can use nnzColwiseM when reserving space for B.
21    // Use M.valuePtr() and M.innerIndexPtr() to define arrays similar
22    // to JM. Use those arrays to access the non-zero entries of M.
23
24    // START
25
26    Eigen::VectorXi nnzColwiseB;
27    // We have that nnz(B.col(j*n+l)) = nnz(M.col(j)) * nnz(M.col(l)).
28    Eigen::MatrixXi tmp = nnzColwiseM * nnzColwiseM.transpose(); // O(n^2)
29    nnzColwiseB = Eigen::MatrixXi::Map(tmp.data(), n * n, 1);
30
31    // Reserve sufficient space.
32    B.reserve(nnzColwiseB);
33
34    // We have that B.block(i*n, j*n, n, n) = M(i, j) * M,
35    // i.e. B( i*n + k, j*n + l ) = M(i, j) * M(k, l).
36    // Hence, we loop over i, j, k, l.
37
38    // Row indices of non-zero entries (size = nnz):
39    const int *IM = M.innerIndexPtr();
40    // Values of non-zero entries (size = nnz):
41    const double *valM = M.valuePtr();
42
43    // Assuming nnz in each column is bounded by a constant, the below is O(n^2).
44    // Loop over the column index j:
45    for (int j = 0; j < n; j++) {
46        // Loop over the non-zero entries in column j:
47        for (int s = JM[j]; s < JM[j + 1]; s++) {
48            int i = IM[s]; // Row index of non-zero entry #s.
49            double Mij = valM[s]; // M(i, j) = Mij
50
51            // We now compute Mij * M.
52            // Loop over the column index l:
53            for (int l = 0; l < n; l++) {
54                // Loop over the non-zero entries in column l:
55                for (int t = JM[l]; t < JM[l + 1]; t++) {
56                    int k = IM[t]; // Row index of non-zero entry #t.
57                    double Mkl = valM[t]; // M(k, l) = Mkl

```