

C++ code 3.11.5: Sub-problem (3-11.e): Addition and recompression of two low-rank matrices

```
2  MatrixLowRank &MatrixLowRank::addTo(const MatrixLowRank &X, double rtol,
3                                     double atol) {
4      assert(_m == X._m && _n == X._n && "Matrices must have the same shape!");
5      assert((rank() + X.rank() <= std::min(_n, _m)) && "Sum of ranks too large!");
6      assert(rtol > 0 && atol > 0 && "Tolerances must be positive!");
7      // TODO: (3-11.e) Implement the function s.t. the  $m \times n$ -matrix stored
8      // in this class is replaced by  $\text{trunc\_tol}(M + X)$ .
9      // Note: X is not an eigen matrix but a matrix of type MatrixLowRank.
10     // Note: We have  $\text{this} = \_A * \_B.\text{transpose}()$ 
11     // Note: We have  $X = X.\_A * X.\_B.\text{transpose}()$ 
12     // Note: This function should be efficient.
13
14     // START
15     const unsigned int r = _r, s = X._r;
16     // We will have  $X + \text{this} = A\tilde{} * B\tilde{}.\text{transpose}()$ 
17     Eigen::MatrixXd Atilde(_m, r + s), Btilde(_n, r + s);
18     Atilde << _A, X._A;
19     Btilde << _B, X._B;
20
21     // Calculate the QR decomposition of Atilde and Btilde.
22     Eigen::HouseholderQR<Eigen::MatrixXd> qrA(Atilde);
23     Eigen::HouseholderQR<Eigen::MatrixXd> qrB(Btilde);
24     Eigen::MatrixXd RA = qrA.matrixQR().triangularView<Eigen::Upper>();
25     Eigen::MatrixXd RB = qrB.matrixQR().triangularView<Eigen::Upper>();
26
27     // We need the full SVD of  $RR = RA * RB.\text{transpose}()$ 
28     Eigen::JacobiSVD<Eigen::MatrixXd> svdRR(
29         RA * RB.transpose(), Eigen::ComputeFullU | Eigen::ComputeFullV);
30
31     // Rank of the sum  $X + \text{this}$  is the (numerical) rank of  $RA * RB.\text{transpose}()$ .
32     Eigen::VectorXd Sigma = svdRR.singularValues();
33     const double sigma1 = Sigma[0];
34     // Take care of the case  $\text{sigma1} = 0$ , i.e.  $X + \text{this} = 0$ .
35     // Follow the convention for storing the zero matrix
36     if (sigma1 < atol) {
37         _r = 1;
38         _A = Eigen::MatrixXd::Zero(_m, 1);
39         _B = Eigen::MatrixXd::Zero(_n, 1);
40
41         return *this;
42     }
43
44     unsigned int rtilde = 1;
45     // Numerical rank is at least 1.
46     while (rtilde < Sigma.size() && Sigma[rtilde] > atol &&
47           Sigma[rtilde] > rtol * sigma1) {
48         // Numerical rank is at least rtilde+1.
49         rtilde++;
50     }
51
52     _r = rtilde;
53     _A = qrA.householderQ() * svdRR.matrixU().leftCols(rtilde) *
54         Sigma.head(rtilde).asDiagonal();
55     _B = qrB.householderQ() * svdRR.matrixV().leftCols(rtilde);
56     // END
```