```
#include <Eigen/Core>
#include <Eigen/Array>
#include <math.h>
#include <time.h>
#include <stdio.h>

// From Eigen FAQ: On the x86 platform, SSE2 is not enabled
// by default; you need to pass the -msse2 compiler option:
// !g++ -O2 -msse2 -I /usr/local/eigen/ coord.cpp -o coord

// Import the most common Eigen types:
USING_PART_OF_NAMESPACE_EIGEN

// This should turn off range checking:
#define EIGEN_NO_DEBUG

void coordinate_descent(MatrixXf A, MatrixXf Y, MatrixXi & S) {

  // Determine basis/data sizes:
  int num_units = A.rows();
  int num_bases = A.cols();
  int num_cases = Y.cols();

  // Specify maximum iterations:
  int max_iter = 128;

  // Specify sparsity parameter:
  float gamma = 0.95000;

  // Specify progress tolerance:
  float tolerance = 0.000001;

  // Set data to have zero mean:
  float num_elements = Y.cols() * Y.rows();
  Y.cwise() -= Y.sum() / num_elements;

  // Set to have unit variance:
  float squared_norm = Y.squaredNorm();
  if ( squared_norm > 0 )
    Y /= sqrt(squared_norm / num_elements);

  // Truncate any outliers:
  float outlier_threshold = 2.0;
  for (int j = 0; j < Y.cols(); ++j)
    for (int i = 0; i < Y.rows(); ++i)
      if (Y(i, j) > outlier_threshold)
        Y(i, j) = outlier_threshold;
      else
        if (Y(i, j) < -outlier_threshold )
          Y(i, j) = -outlier_threshold;

  // Scale to the unit interval:
  Y /= 2 * outlier_threshold;
  Y.cwise() += 0.5;

  // Initialize the matrix AtA:
  MatrixXf AtA = A.transpose() * A;
```

```
// Number of alphas is determined at compile time:
int num_alphas = 5;
MatrixXf alphas(1,num_alphas);
// Initialize gradient steps:
alphas << 1.0000, 0.3000, 0.1000, 0.0300, 0.0100;

// Number of steps is determined at compile time:
MatrixXf alphas_plus_zero(1,num_alphas + 1);
alphas_plus_zero << alphas, 0.000;

// Used track the step indices:
MatrixXi alphaIndex(1,num_cases);

// Index indicating no progress:
int no_progress = num_alphas;

// Used to replicate alpha steps:
MatrixXf alpha_replicator = MatrixXf::Ones(num_bases,1);

// Vector of diagonal elements:
MatrixXf P = (AtA.diagonal()).transpose();

// Used to replicate diagonal elements:
MatrixXf diagonal_replicator = MatrixXf::Ones(num_cases,1);

// Replicate the diagonal elements:
MatrixXf P_replicated = diagonal_replicator * P;

// Initialize the Matrix YtA:
MatrixXf YtA = Y.transpose() * A;

// Initialize the coefficient matrix:
MatrixXf X = MatrixXf::Zero(num_cases,num_bases);

// Intermediate matrix results:
MatrixXf Y_minus_Ax_t_A(num_units,num_cases);
MatrixXf Q(num_units,num_cases);
MatrixXf delta(num_units, num_cases);
MatrixXf DtAtA(num_units,num_cases);

// Terms used in the line searches:
MatrixXf av(num_cases,1);
MatrixXf bv(num_cases,1);
MatrixXf minHx(num_cases,1);
MatrixXf Hx(num_cases,1);
MatrixXf Xn(num_cases,num_bases);

// Create vector of selected steps:
MatrixXf best_alphas(1, num_cases);

for (int iter = 0; iter < max_iter; ++iter) {

  // Compute the gradient vector:
  Y_minus_Ax_t_A = YtA - X * AtA;
  Q = Y_minus_Ax_t_A + P_replicated.cwise() * X;

  for (int i = 0; i < Q.rows(); ++i)
    for (int j = 0; j < Q.cols(); ++j)
```

```
      if (fabs(Q(i,j)) < gamma)
        Q(i,j) = 0.0;
      else if ( Q(i,j) < 0 )
        Q(i,j) += gamma;
      else
        Q(i,j) -= gamma;

  // Now compute the derivative:
  delta = (Q.cwise() / P_replicated) - X;

  // Compute terms for line search:

  DtAtA = delta * AtA.transpose();

  // Compute a = 0.5 * d' * AtA * d:
  av = ((DtAtA.cwise() * delta) * 0.5).rowwise().sum();

  // Compute b = - y_minus_Ax_t_A*d:
  bv = (Y_minus_Ax_t_A.cwise() * delta * -1.0).rowwise().sum();

  // Find the minimizing step size:
  minHx = gamma * X.cwise().abs().rowwise().sum();

  // Restore the best-step indices:
  alphaIndex.setConstant(no_progress);

  // Carry out the line search:
  for (int k = 0; k < num_alphas; ++k) {
    float alpha = alphas(k);
    Xn = X + delta * alpha;
    Hx = av * alpha * alpha +
            bv * alpha + Xn.cwise().abs().rowwise().sum() * gamma;
    for (int i = 0; i < num_cases; ++i) {
      if (Hx(i,0) < (minHx(i,0) * (1 - tolerance))) {
        alphaIndex(0,i) = k;
      }
    }
    minHx = minHx.cwise().min(Hx);
  }

  // Another place indexing would help:
  for (int i = 0; i < num_cases; ++i) {
    best_alphas(0,i) = alphas_plus_zero(alphaIndex(i));
  }

  // Apply the gradient-step update:
  X += (alpha_replicator * best_alphas).transpose().cwise() * delta;
}
MatrixXf Z = X.transpose();
// Issue some indication of completion:
if ( Z.rows() > 10 || Z.cols() > 10 ) {
  std::cout << "Coefficient matrix too large:" << '\n';
  std::cout << Z(0, 0) << '\n' << '\n';
} else {
  std::cout << "Transposed coefficient matrix:" << '\n';
  std::cout << Z << '\n' << '\n';
}
// Compute the sum of non-zero coefficients:
  S = (X.cwise() != 0.0).cast<int>().colwise().sum();

  std::cout << S << '\n' << '\n';
}


int main(int, char *[]) {

  // Specify the basis and data size:
  int num_units = 3;
  int num_bases = 5;
  int num_cases = 7;

  // Initialize basis vector matrix:
  MatrixXf A(num_units, num_bases);
  A << 0.8147, 0.9134, 0.2785, 0.9649, 0.9572,
       0.9058, 0.6324, 0.5469, 0.1576, 0.4854,
       0.1270, 0.0975, 0.9575, 0.9706, 0.8003;

  // Initialize the data vector matrix;
  MatrixXf Y(num_units, num_cases);
  Y << 0.1419, 0.7922, 0.0357, 0.6787, 0.3922, 0.7060, 0.0462,
       0.4218, 0.9595, 0.8491, 0.7577, 0.6555, 0.0318, 0.0971,
       0.9157, 0.6557, 0.9340, 0.7431, 0.1712, 0.2769, 0.8235;

  //  int num_units = 845;  // 13 * 13 * 5 = 845
  //  int num_bases = 384;  // 256  *  1.5 = 384
  //  int num_cases = 1000;

  //  MatrixXf A = MatrixXf::Random(num_units, num_bases);
  //  MatrixXf Y = MatrixXf::Random(num_units, num_cases);

  // 845 ; 256 ; 1000 => Elapsed time is 15.730947 seconds.
  // 845 ; 384 ; 1000 => Elapsed time is 31.783866 seconds.

  // Declare return matrix:
  MatrixXi S;

  // Start the CPU timer:
  clock_t start = clock();

  // Run coordinate descent:
  coordinate_descent(A, Y, S);

  // Elapsed time in seconds:
  printf("Elapsed time is %.6f seconds\n",
          ((double)clock() - start) / CLOCKS_PER_SEC);

  // Print small descriptors:
  if ( S.cols() < 10 ) {
    std::cout << "Sum of non-zero coefficients:" << '\n';
    std::cout << S << '\n' << '\n';
  }
}
```