```cpp
void cHMM::trainMS(std::vector< std::vector<Sample_3d> > trainingset){

    boost::numeric::ublas::matrix<double> A_up(numStati, numStati);
    boost::numeric::ublas::matrix<double> A_down(numStati, numStati);

    std::vector<Gaussian_3d_mixture> gaussians_up;
    std::vector<Gaussian_3d_mixture> gaussians_down;

    // inizializza A_up, A_down
    for(int i=0; i<numStati; i++){
        for(int j=0; j<numStati; j++){
            A_up(i,j) = 0;
            A_down(i,j) = 0;
        }
    }

    // inizializa gaussians_up, gaussians_down con tutti zero
    for(int k=0; k<numStati; k++){

        Gaussian_3d_mixture* g1 = new Gaussian_3d_mixture(nGauss, true);
        Gaussian_3d_mixture* g2 = new Gaussian_3d_mixture(nGauss, true);

        gaussians_up.push_back( *g1 );
        gaussians_down.push_back( *g2 );

    }

    for(int data=0; data<trainingset.size(); data++){

        std::vector<Sample_3d> current = trainingset.at(data);
        double scale[current.size()];
        boost::numeric::ublas::matrix<double> alpha(numStati, current.size());
        boost::numeric::ublas::matrix<double> beta(numStati, current.size());
        forwardProc_scale(current, alpha, scale);
        backwardProc(current, beta, scale);

        // debug
        /*
         std::cout<< "scale:" << std::endl;
         for(int d=0; d<current.size(); d++)
             std::cout<<scale[d]<<std::endl;
         */

        double P = getProbabilityFromScale(scale, current.size());

        //debug
        std::cout<<"Probabilità: "<<P<<std::endl;

        // aggiornamento pi
        if(isErgodic){
            for(int i=0; i<numStati; i++)
                pi[i] = alpha(i,1) * beta(i,1) / P;
        }

        // aggiornamento A
        for(int i=0; i<numStati; i++){

            for(int j=0; j<numStati; j++){

                double up = 0;
                double down = 0;

                for(int t=0; t<current.size()-2; t++){

                    up += alpha(i,t) * A(i,j) * B(j,current.at(t+1)) * beta(j,t+1);
                    down += alpha(i,t) * beta(j,t);

                }//t

                A_up(i,j) += up;// / P;
                A_down(i,j) += down;// / P;

            }//j
        }//i

        //debug
        //std::cout<<A_up<<std::endl;
        //std::cout<<A_down<<std::endl;

        // aggiornamento parametri gaussiane
        // gamma[t][j][k]
        double ***gamma;
        gamma = (double ***)malloc(current.size() * sizeof(double **));
        for (int t = 0; t < current.size(); t++)
        {
            gamma[t] = (double **)malloc(numStati * sizeof(double *));
            for (int j = 0; j < numStati; j++)
            {
            gamma[t][j] = (double *)malloc(mixture_vect.at(j).howmany * sizeof(double));
            }
        }

        // calcolo gamma
        for(int t=0; t<current.size(); t++){ // ciclo sui sample della gesture

            double sum = 0;
            for(int j=0; j<numStati; j++){
                sum += alpha(j,t)*beta(j,t);
            }

            for(int j=0; j<numStati; j++){ // ciclo sugli stati

                for(int k=0; k<mixture_vect.at(j).howmany; k++){ // ciclo sulle componenti della mixture dello stato corrente

                    gamma[t][j][k] = alpha(j,t) * beta(j,t) * mixture_vect.at(j).weight[k] *
                                       mixture_vect.at(j).components.at(k).pdf_3d(current.at(t)) /
                                       ( sum * B(j, current.at(t)) );

                }
            }

        }


        // aggiornamento pesi misture: mixture_vect.at(STATO).weight[MISTURA]
        for(int j=0; j<numStati; j++){

            int n_mix = mixture_vect.at(j).howmany;

            for(int k=0; k<n_mix; k++){

                double up = 0, down = 0;

                for(int t=0; t<current.size(); t++){

                    up += gamma[t][j][k];

                    for(int m=0; m<n_mix; m++){

                        down += gamma[t][j][m];
                    }

                }

                gaussians_up.at(j).weight[k] += up;// / P;
                gaussians_down.at(j).weight[k] += down;// / P;

                //mixture_vect.at(j).weight[k] = up / down;

            }
        }

        // aggiornamento medie
        for(int j=0; j<numStati; j++){

            int n_mix = mixture_vect.at(j).howmany;

            for(int k=0; k<n_mix; k++){

                double down = 0;

                for(int t=0; t<current.size(); t++){

                    down += gamma[t][j][k];

                }

                for(int n=0; n<3; n++){

                    double up = 0;

                    for(int t=0; t<current.size(); t++){

                        up += gamma[t][j][k] * current.at(t)[n];

                    }

                    gaussians_up.at(j).components.at(k).mean[n] += up;// / P;
                    gaussians_down.at(j).components.at(k).mean[n] += down;// / P;

                    //mixture_vect.at(j).components.at(k).mean[n] = up / down;

                }
            }
        }

        // aggiornamento covarianze
        for(int j=0; j<numStati; j++){

            int n_mix = mixture_vect.at(j).howmany;

            for(int k=0; k<n_mix; k++){

                double down = 0;

                for(int t=0; t<current.size(); t++){

                    down += gamma[t][j][k];

                }

                for(int n=0; n<3; n++){

                    double up = 0;

                    for(int t=0; t<current.size(); t++){

                        up += gamma[t][j][k] *
                                (current.at(t)[n] - mixture_vect.at(j).components.at(k).mean[n]) *
                                (current.at(t)[n] - mixture_vect.at(j).components.at(k).mean[n]);

                    }

                    // nota: si aggiornano solo le covarianze sulla diagonale, in quanto lavoriamo
                    // con matrici di covarianza diagonali
                    gaussians_up.at(j).components.at(k).cov(n,n) += up;// / P;
                    gaussians_down.at(j).components.at(k).cov(n,n) += down;// / P;

                    //mixture_vect.at(j).components.at(k).cov(n,n) = up / down;

                }//n
            }//k
        }//j

        // può servire check su covarianze, cioè se cov(i,i) < K, allora cov(i,i) = K


        // libera la memoria
        for (int t = 0; t < current.size(); t++){
            for (int j = 0; j < numStati; j++){
                free(gamma[t][j]);
            }
            free(gamma[t]);
        }
        free(gamma);

    }//data

    // aggiorna le matrici A e tutti i parametri delle gaussiane
    for(int i=0; i<numStati; i++){ // cicla sugli stati

        // aggiorna la matrice A
        for(int j=0; j<numStati; j++)
            A(i,j) = A_up(i,j) / A_down(i,j);

        // aggiorna i parametri delle gaussiane
        for(int k=0; k<mixture_vect.at(i).howmany; k++){

            // aggiorna i pesi
            mixture_vect.at(i).weight[k] = gaussians_up.at(i).weight[k] / gaussians_down.at(i).weight[k];

            for(int n=0; n<3; n++){ // cicla sulle dimensioni della gaussiana (3D)

                // aggiorna le medie
                mixture_vect.at(i).components.at(k).mean[n] = gaussians_up.at(i).components.at(k).mean[n] / gaussians_down.at(i).components.at(k).mean[n];

                // aggiorna le covarianze
                // nota: si aggiornano solo quelle sulla diagonale, in quanto si lavora, per ipotesi, con matrici diagonali
                mixture_vect.at(i).components.at(k).cov(n,n) = gaussians_up.at(i).components.at(k).cov(n,n) / gaussians_down.at(i).components.at(k).cov(n,n);
            }

        }

    }

    //debug
    std::cout<<"Fine trainMS"<<std::endl;
}
```