```cpp
#include "NelderMead.h"

NelderMead::NelderMead(SegmentData &segment)
{
  generateSimplex();
  countAllMetrics(segment);
  sort();
  calculate(segment);
}

void NelderMead::calculate(SegmentData &segment){
  double current_best_metric = DBL_MAX;
  double Xg_p = 0, Xg_cg = 0, Xg_c = 0, Xg_dt = 0, Xg_h = 0, Xg_k = 0;
  int size_without_last = _simplex.size() - 1;
  int last_simplex_index = size_without_last;

  for (int x = 0; x < ITERATIONS; x++){
    //set new best metric
    current_best_metric = min(current_best_metric, _simplex[0].getMetric());

    // Termination criteria
    if (current_best_metric < METRIC_EPSILON) {
      cout << "Found metric " << current_best_metric << " less then epsilon " << METRIC_EPSILON << endl;
      break;
    }

    // -- REFLECTION -- //
    for (int z = 0; z < size_without_last; z++){ //Xg - sou□et v□ech a□ na posledn□ (nejhor□□)
      Xg_p += _simplex[z].getP();
      Xg_cg += _simplex[z].getCg();
      Xg_c += _simplex[z].getC();
      Xg_dt += _simplex[z].getDt();
      Xg_h += _simplex[z].getH();
      Xg_k += _simplex[z].getK();
    }
    // Xg //
    Xg_p /= size_without_last;
    Xg_cg /= size_without_last;
    Xg_c /= size_without_last;
    Xg_dt /= size_without_last;
    Xg_h /= size_without_last;
    Xg_k /= size_without_last;
    // Xr //
    EquationData Xr(
      (A*(Xg_p - _simplex[last_simplex_index].getP())) + _simplex[last_simplex_index].getP(),
      (A*(Xg_cg - _simplex[last_simplex_index].getCg())) + _simplex[last_simplex_index].getCg(),
      (A*(Xg_c - _simplex[last_simplex_index].getC())) + _simplex[last_simplex_index].getC(),
      (A*(Xg_dt - _simplex[last_simplex_index].getDt())) + _simplex[last_simplex_index].getDt(),
      (A*(Xg_h - _simplex[last_simplex_index].getH())) + _simplex[last_simplex_index].getH(),
      (A*(Xg_k - _simplex[last_simplex_index].getK())) + _simplex[last_simplex_index].getK()
    );
    Xr.setMetric(countMetrics(Xr, segment));
    // Reflection condition: YES => update the worst parameters; NO => contraction or expansion //
    if (_simplex[0].getMetric() < Xr.getMetric() && Xr.getMetric() < _simplex[last_simplex_index - 1].getMetric()){
```

```
   _simplex[last_simplex_index] = Xr;
  }
  // Contraction/Expansion condition: YES => Contraction; NO => Expansion //
  else if (_simplex[0].getMetric() < Xr.getMetric()){
   // -- CONTRACTION -- //
   // Xc //
   EquationData Xc(
    (G*(_simplex[last_simplex_index].getP() - Xg_p)) + Xg_p,
    (G*(_simplex[last_simplex_index].getCg() - Xg_cg)) + Xg_cg,
    (G*(_simplex[last_simplex_index].getC() - Xg_c)) + Xg_c,
    (G*(_simplex[last_simplex_index].getDt() - Xg_dt)) + Xg_dt,
    (G*(_simplex[last_simplex_index].getH() - Xg_h)) + Xg_h,
    (G*(_simplex[last_simplex_index].getK() - Xg_k)) + Xg_k
   );
   Xc.setMetric(countMetrics(Xc, segment));
   // Contraction condition: YES => update the worst parameters; NO => multiple contraction //
   if (Xc.getMetric() < _simplex[last_simplex_index].getMetric()){
    _simplex[last_simplex_index] = Xc;
   }
   else {
    // -- MULTIPLE CONTRACTION-- //
    EquationData tmp_eq;
    for (int size = _simplex.size(), k = 1; k < size; k++){ // contraction all sets of parameters without the best one
     tmp_eq.setAllParameters(
      (H*(_simplex[k].getP() - _simplex[0].getP())) + _simplex[0].getP(),
      (H*(_simplex[k].getCg() - _simplex[0].getCg())) + _simplex[0].getCg(),
      (H*(_simplex[k].getC() - _simplex[0].getC())) + _simplex[0].getC(),
      (H*(_simplex[k].getDt() - _simplex[0].getDt())) + _simplex[0].getP(),
      (H*(_simplex[k].getH() - _simplex[0].getH())) + _simplex[0].getH(),
      (H*(_simplex[k].getK() - _simplex[0].getK())) + _simplex[0].getK()
     );
     _simplex[k] = tmp_eq;
    }
   }
  }
  else {
   // -- EXPANSION -- //
   // Xe //
   EquationData Xe(
    (B*(Xr.getP() - Xg_p)) + Xr.getP(),
    (B*(Xr.getCg() - Xg_cg)) + Xr.getCg(),
    (B*(Xr.getC() - Xg_c)) + Xr.getC(),
    (B*(Xr.getDt() - Xg_dt)) + Xr.getDt(),
    (B*(Xr.getH() - Xg_h)) + Xr.getH(),
    (B*(Xr.getK() - Xg_k)) + Xr.getK()
   );
   Xe.setMetric(countMetrics(Xe, segment));
   // Expansion condition: ALWAYS => update the worst parameters; YES => update by Xe; NO => update by Xr //
   if (Xe.getMetric() < Xr.getMetric()){
    _simplex[last_simplex_index] = Xe;
   }
   else {
    _simplex[last_simplex_index] = Xr;
   }
  }
  sort();
 }
 // Save the best parameters into segment //
 segment.setResult(_simplex[0]);
}
```

```cpp

void NelderMead::generateSimplex(){
  default_random_engine rnd((unsigned) chrono::system_clock::now().time_since_epoch().count());

  uniform_real_distribution<double> gen_p(EquationBounds::pmin, EquationBounds::pmax);
  uniform_real_distribution<double> gen_cg(EquationBounds::cgmin, EquationBounds::cgmax);
  uniform_real_distribution<double> gen_c(EquationBounds::cmin, EquationBounds::cmax);
  uniform_real_distribution<double> gen_dt(EquationBounds::dtmin, EquationBounds::dtmax);
  uniform_real_distribution<double> gen_h(EquationBounds::hmin, EquationBounds::hmax);
  uniform_real_distribution<double> gen_k(EquationBounds::kmin, EquationBounds::kmax);

  if (!_simplex.empty()){
    cout << "Clear simplex before generate new one..." << endl;
    _simplex.clear();
  }

  EquationData eq;
  for (int i = 0; i < NUM_OF_PARAM_SETS; i++){
    eq.setAllParameters(gen_p(rnd), gen_cg(rnd), gen_c(rnd), gen_dt(rnd), gen_h(rnd), gen_k(rnd));
    _simplex.push_back(eq);
  }
}

void NelderMead::countAllMetrics(SegmentData &segment){
  double metric;
  bool least_one_good = false;
  for (int size = _simplex.size(), i = 0; i < size; i++){
    metric = countMetrics(_simplex[i], segment);
    _simplex[i].setMetric(metric);
    if (metric != DBL_MAX){
      least_one_good = true;
    }
  }
  if (!least_one_good){
    cout << "Any metric isnt good => generate new simplex..." << endl;
    generateSimplex();
    countAllMetrics(segment);
  }
}

double NelderMead::countMetrics(EquationData &params, SegmentData &segment){
  double current_blood_time;
  double eq_fi, eq_alfa, eq_beta, eq_gama, eq_D, eq_b;
  double relative_error, total_relative_error, sum_relative_error = 0, deviation = 0, sum_deviation = 0;
  int blood_values_count = segment.getBloodValues().size();
  int total_count_metrics_values = blood_values_count;
  for (int i = 0; i < blood_values_count; i++){
    current_blood_time = segment.getBloodValues()[i].getDate();

    eq_fi = current_blood_time + params.getDt() + (params.getK() * ((segment.getInterpolationIst(current_blood_time) - segment.getInterpolationIst(current_blood_time - params.getH())) / params.getH()));
    if (eq_fi < segment.getMinDate() || eq_fi > segment.getMaxDate()){
      total_count_metrics_values--;
      continue;
    }
    eq_alfa = params.getCg();
    eq_beta = params.getP() - (eq_alfa * segment.getInterpolationIst(current_blood_time));
    eq_gama = params.getC() - segment.getInterpolationIst(eq_fi);
    eq_D = pow(eq_beta, 2) - (4 * eq_alfa*eq_gama);
    if (eq_D < 0.0) eq_D = 0.0;
    eq_b = (-eq_beta + sqrt(eq_D)) / (2 * eq_alfa);

    relative_error = abs(segment.getBloodValues()[i].getBlood() - eq_b) / segment.getBloodValues()[i].getBlood();
    sum_relative_error += relative_error;
    sum_deviation += pow(relative_error, 2);
  }


  if (total_count_metrics_values == 0){
    return DBL_MAX;
  }
  else {
    total_relative_error = sum_relative_error / total_count_metrics_values;
    deviation = sqrt((sum_deviation / total_count_metrics_values) - pow(total_relative_error, 2));
    return total_relative_error + deviation;
  }
}

void NelderMead::sort(){
  std::sort(_simplex.begin(), _simplex.end());
}
```