```
 1: #ifndef _CONFIG_H
 2: #define _CONFIG_H
 3:
 4: //if this header is included, program will run with Intel TBB
 5: #include "tbb.h"
 6:
 7: // If debug set, exit will wait for Enter pressed
 8: #define DEBUG 1
 9:
10: #define TIME_MEASURING
11:
12: //the number of iteration for Nelder-Mead optimalization
13: #define ITERATIONS 10000
14: //if metric is less then this epsilon Nelder-Mead optimalization will be terminate
15: #define METRIC_EPSILON 5e-5
16: //the number of simplex parameters sets
17: #define NUM_OF_PARAM_SETS 50
18:
19: #endif
```

```
 1: #ifndef _CONSTANTS_H
 2: #define _CONSTANTS_H
 3:
 4: //Nealder-Mead algoritmus constants
 5: #define A 1.0
 6: #define B 1.0
 7: #define G 0.5
 8: #define H 0.5
 9:
10: // Equation default bounds
11: #define P_MAX 2.0
12: #define P_MIN 0.0
13: #define CG_MAX 0.0
14: #define CG_MIN -0.5
15: #define C_MAX 10.0
16: #define C_MIN -10.0
17: #define DT_MAX 0.0277777
18: #define DT_MIN 0.0
19: #define H_MAX 0.0277777
20: #define H_MIN 0.0
21: #define K_MAX 1.0
22: #define K_MIN -1.0
23:
24: #endif
```

```
 1: #include "main.h"
 2:
 3: using namespace std;
 4: using namespace std::chrono;
 5:
 6: int main(int argc, char **argv) {
 7: #ifdef TIME_MEASURING
 8:   cout << "Time measuring ON" << endl;
 9:   high_resolution_clock::time_point full_start, full_end, alg_start, alg_end;
10:   full_start = high_resolution_clock::now();
11: #endif
12:   vector<SegmentData> segments;
13:   CmdDriver parser(argc, argv);
14:
15:   std::atexit(exit_handler);
16:
17:   SqliteDriver sql(parser.getParam("sql"));
18:   FileDriver file(parser.getParam("bounds"));
19:
20:   try {
21:     segments = sql.load(segments);
22:     file.load();
23:   }
24:   catch (string& msg){
25:     cerr << "Load file problem!\n" << msg << endl;
26:     return(EXIT_FAILURE);
27:   }
28:
29:   EquationBounds::printEquationBounds();
30:
31: #ifdef TIME_MEASURING
32:   alg_start = high_resolution_clock::now();
33: #endif
34: #ifdef _INTEL_TBB_H
35:   cout << "Run with Intel TTB" << endl;
36:   int size = segments.size();
37:   tbb::task_scheduler_init init(4);
38:   tbb::parallel_for(0, size, [&](int i){
39:     NelderMead alg(segments[i]);
40:     cout << "Segment " << std::setfill(' ') << std::setw(3) <<
    segments[i].getId() << " " << segments[i].getResult() << endl;
41:   });
42: #else
43:   cout << "Run without Intel TTB" << endl;
44:   for (int size = segments.size(), i = 0; i < size; i++) {
45:     NelderMead alg(segments[i]);
46:     cout << "Segment " << std::setfill(' ') << std::setw(3) <<
    segments[i].getId() << " " << segments[i].getResult() << endl;
47:   }
48: #endif
49: #ifdef TIME_MEASURING
50:   alg_end = high_resolution_clock::now();
51: #endif
52:
53:   try {
54:     sql.save(segments);
55:   }
56:   catch (string& msg){
57:     cerr << "Save SQLite problem!\n" << msg << endl;
58:   }
59:
60: #ifdef TIME_MEASURING
61:   full_end = high_resolution_clock::now();
62:   auto full_duration = duration_cast<microseconds>(full_end - full_start).count();
```

```
63:   auto alg_duration = duration_cast<microseconds>(alg_end - alg_start).count();
64:
65:   cout << "Time execute for Nelder-Mead algorithm: " << alg_duration << "us => "
    << alg_duration / 1000.0 << "ms => " << alg_duration / 1000000.0 << "s" <<  endl;
66:   cout << "Time execute for the entire program: " << full_duration << "us => " <<
    alg_duration / 1000.0 << "ms => " << alg_duration / 1000000.0 << "s" << endl;
67: #endif
68:
69:   return(EXIT_SUCCESS);
70: }
71:
72: void exit_handler(){
73:   if (DEBUG){
74:     std::cout << "Press Enter to exit..." << std::endl;
75:     std::cin.get();
76:   }
77: }
```

```
 1:
 2: #include "main.h"
 3: #include <sstream>
 4:
 5: #define MPI_TASK_ID_TAG 1
 6: #define MPI_RESULT_TAG 2
 7: #define RESULT_VECTOR_SIZE 8
 8:
 9: using namespace std;
10: using namespace std::chrono;
11:
12: int segment_count;
13: int NOHTING_TO_DO = -666;
14: int WORK_DONE = 666;
15:
16:
17: int main(int argc, char **argv) {
18: #ifdef TIME_MEASURING
19:   cout << "Time measuring ON" << endl;
20:   high_resolution_clock::time_point full_start, full_end, alg_start, alg_end;
21:   full_start = high_resolution_clock::now();
22: #endif
23:   int mpi_node_id, mpi_process_count;
24:   vector<SegmentData> segments, computed_segments;
25:
26:   std::atexit(exit_handler);
27:
28:     CmdDriver parser(argc, argv);
29:   SqliteDriver sql(parser.getParam("sql"));
30:   FileDriver file(parser.getParam("bounds"));
31:
32:   try {
33:     segments = sql.load(segments);
34:     file.load();
35:   }
36:   catch (string& msg){
37:     cerr << "Load file problem!\n" << msg << endl;
38:     return(EXIT_FAILURE);
39:   }
40:   segment_count = segments.size();
41:
42:   MPI_Init(&argc, &argv);
43:   MPI_Comm_rank(MPI_COMM_WORLD, &mpi_node_id);
44:   MPI_Comm_size(MPI_COMM_WORLD, &mpi_process_count);
45:
46:
47:   if (mpi_node_id == MASTER_ID) {
48:     EquationBounds::printEquationBounds();
49:     cout << "Master start (" << mpi_node_id << ")" << endl;
50:     master(mpi_process_count, segments);
51:   }
52:   else {
53:     cout << "Slave start (" << mpi_node_id << ")" << endl;
54: #ifdef TIME_MEASURING
55:   alg_start = high_resolution_clock::now();
56: #endif
57:     slave(mpi_node_id, segments);
58: #ifdef TIME_MEASURING
59:   alg_end = high_resolution_clock::now();
60: #endif
61:     cout << endl << "================== RESULT from " << mpi_node_id << "
    ==================" << endl;
62:     int size = segments.size();
63:     for(int i = 0; i < size; i++){
```

```
64:       if(segments[i].getResult().getMetric() != DBL_MAX){
65:         computed_segments.push_back(segments[i]);
66:         // cout << mpi_node_id << " Segment " << segments[i].getId() << ": " <<
    segments[i].getResult() << endl;
67:       }
68:     }
69:     size = computed_segments.size();
70:     for(int i = 0; i < size; i++){
71:       cout << mpi_node_id << " Segment " << computed_segments[i].getId() << ": "
    << computed_segments[i].getResult() << endl;
72:     }
73:     cout << endl << "====================================" << endl << endl;
74:     try {
75:       sql.save(computed_segments);
76:     }
77:     catch (string& msg){
78:       cerr << mpi_node_id << ": Save SQLite problem!\n" << msg << endl;
79:     }
80:   }
81:
82:   MPI_Finalize();
83:
84: #ifdef TIME_MEASURING
85:   full_end = high_resolution_clock::now();
86:   auto full_duration = duration_cast<microseconds>(full_end -
    full_start).count();
87:   auto alg_duration = duration_cast<microseconds>(alg_end - alg_start).count();
88:
89:   cout << "#" << mpi_node_id << "Time execute for Nelder-Mead algorithm: " <<
    alg_duration << "us => " << alg_duration / 1000.0 << "ms => " << alg_duration /
    1000000.0 << "s" <<  endl;
90:   cout << "#" << mpi_node_id << "Time execute for the entire program: " <<
    full_duration << "us => " << alg_duration / 1000.0 << "ms => " << alg_duration /
    1000000.0 << "s" << endl;
91: #endif
92:
93:   return(EXIT_SUCCESS);
94: }
95:
96: void master(int mpi_process_count, vector<SegmentData> &segments){
97:   string msg_head = "#0: ";
98:   cout << msg_head << "I'm MPI master!" << endl;
99:   MPI_Status status;
100:   int remaining_segments_count = segment_count;
101:   int received_responses = 0;
102:
103:   int current_segment_index = 0;
104:   //kazdemu procesu poslu index segmentu, ktery ma zpracovat
105:   for (int i = 1; i < mpi_process_count; i++, current_segment_index++,
    remaining_segments_count--) {
106:     MPI_Send(&current_segment_index, 1, MPI_INT, i, MPI_TASK_ID_TAG,
    MPI_COMM_WORLD);
107:     // cout << msg_head << "Send work on INDEX " << current_segment_index << "
    with ID " << segments[current_segment_index].getId() << " to " << i << endl;
108:   }
109:
110:   int slave_id = 0;
111:   int result = 0;
112:   while (remaining_segments_count > 0) {
113:     MPI_Recv(&result, 1, MPI_INT, MPI_ANY_SOURCE, MPI_RESULT_TAG, MPI_COMM_WORLD,
     &status);
114:     slave_id = status.MPI_SOURCE;
115:     received_responses++;
116:
```

```
117:     MPI_Send(&current_segment_index, 1, MPI_INT, slave_id, MPI_TASK_ID_TAG,
     MPI_COMM_WORLD);
118:
119:     current_segment_index++;
120:     remaining_segments_count--;
121:   }
122:
123:   while((segment_count - received_responses) > 0) {
124:     MPI_Recv(&result, 1, MPI_INT, MPI_ANY_SOURCE, MPI_RESULT_TAG, MPI_COMM_WORLD,
      &status);
125:     received_responses++;
126:     slave_id = status.MPI_SOURCE;
127:     MPI_Send(&NOHTING_TO_DO, 1, MPI_INT, slave_id, MPI_TASK_ID_TAG,
     MPI_COMM_WORLD);
128:   }
129: }
130:
131: void slave(int my_id, vector<SegmentData> &segments){
132:   ostringstream oss_msg_head;
133:   oss_msg_head << "#" << my_id << ": ";
134:   string msg_head = oss_msg_head.str();
135:
136:   cout << "I'm MPI slave #" << my_id << "..." << endl;
137:   MPI_Status status;
138:   int working_segment = -1;
139:   vector<double> result(RESULT_VECTOR_SIZE);
140:
141:   do {
142:     // cout << msg_head << "Waiting for some job..." << endl;
143:     MPI_Recv(&working_segment, 1, MPI_INT, MASTER_ID, MPI_TASK_ID_TAG,
     MPI_COMM_WORLD, &status);
144:
145:     if (working_segment != NOHTING_TO_DO) {
146:       // cout << msg_head << "I do work on segment " <<
     segments[working_segment].getId() << endl;
147:       // cout << segments[working_segment] << endl;
148:       NealderMead alg(segments[working_segment]);
149:       // cout << msg_head << "My result is " <<
     segments[working_segment].getResult() << endl;
150:       // cout << msg_head << "Best simplex params " << alg.getSimplex()[0] <<
     endl;
151:
152:       MPI_Send(&WORK_DONE, 1, MPI_INT, MASTER_ID, MPI_RESULT_TAG,
     MPI_COMM_WORLD);
153:       // cout << msg_head << "Send result vector" << endl;
154:     }
155:     // else {
156:     //  cout << msg_head << "Nothing to do..." << endl;
157:     // }
158:   } while (working_segment >= 0);
159:   // cout << msg_head << "I'm done with my job!" << endl;
160: }
161:
162:
163: void exit_handler(){
164:   if (DEBUG){
165:     std::cout << "Press Enter to exit..." << std::endl;
166:     std::cin.get();
167:   }
168: }
```

```
 1: #include "NelderMead.h"
 2:
 3: NelderMead::NelderMead(SegmentData &segment)
 4: {
 5:   generateSimplex();
 6:   countAllMetrics(segment);
 7:   sort();
 8:   calculate(segment);
 9: }
10:
11: void NelderMead::calculate(SegmentData &segment){
12:   double current_best_metric = DBL_MAX;
13:   double Xg_p = 0, Xg_cg = 0, Xg_c = 0, Xg_dt = 0, Xg_h = 0, Xg_k = 0;
14:   int size_without_last = _simplex.size() - 1;
15:   int last_simplex_index = size_without_last;
16:
17:   for (int x = 0; x < ITERATIONS; x++){
18:     //set new best metric
19:     current_best_metric = min(current_best_metric, _simplex[0].getMetric());
20:
21:     // Termination criteria
22:     if (current_best_metric < METRIC_EPSILON) {
23:       cout << "Found metric " << current_best_metric << " less then epsilon " <<
    METRIC_EPSILON << endl;
24:       break;
25:     }
26:
27:     // -- REFLECTION -- //
28:     for (int z = 0; z < size_without_last; z++){ //Xg - sou□et v□ech a□ na
    posledn□ (nejhor□□)
29:       Xg_p += _simplex[z].getP();
30:       Xg_cg += _simplex[z].getCg();
31:       Xg_c += _simplex[z].getC();
32:       Xg_dt += _simplex[z].getDt();
33:       Xg_h += _simplex[z].getH();
34:       Xg_k += _simplex[z].getK();
35:     }
36:     // Xg //
37:     Xg_p /= size_without_last;
38:     Xg_cg /= size_without_last;
39:     Xg_c /= size_without_last;
40:     Xg_dt /= size_without_last;
41:     Xg_h /= size_without_last;
42:     Xg_k /= size_without_last;
43:     // Xr //
44:     EquationData Xr(
45:       (A*(Xg_p - _simplex[last_simplex_index].getP())) +
    _simplex[last_simplex_index].getP(),
46:       (A*(Xg_cg - _simplex[last_simplex_index].getCg())) +
    _simplex[last_simplex_index].getCg(),
47:       (A*(Xg_c - _simplex[last_simplex_index].getC())) +
    _simplex[last_simplex_index].getC(),
48:       (A*(Xg_dt - _simplex[last_simplex_index].getDt())) +
    _simplex[last_simplex_index].getDt(),
49:       (A*(Xg_h - _simplex[last_simplex_index].getH())) +
    _simplex[last_simplex_index].getH(),
50:       (A*(Xg_k - _simplex[last_simplex_index].getK())) +
    _simplex[last_simplex_index].getK()
51:     );
52:     Xr.setMetric(countMetrics(Xr, segment));
53:     // Reflection condition: YES => update the worst parameters; NO =>
    contraction or expansion //
54:     if (_simplex[0].getMetric() < Xr.getMetric() && Xr.getMetric() <
    _simplex[last_simplex_index - 1].getMetric()){
```

```
55:       _simplex[last_simplex_index] = Xr;
56:     }
57:     // Contraction/Expansion condition: YES => Contraction; NO => Expansion //
58:     else if (_simplex[0].getMetric() < Xr.getMetric()){
59:       // -- CONTRACTION -- //
60:       // Xc //
61:       EquationData Xc(
62:         (G*(_simplex[last_simplex_index].getP() - Xg_p)) + Xg_p,
63:         (G*(_simplex[last_simplex_index].getCg() - Xg_cg)) + Xg_cg,
64:         (G*(_simplex[last_simplex_index].getC() - Xg_c)) + Xg_c,
65:         (G*(_simplex[last_simplex_index].getDt() - Xg_dt)) + Xg_dt,
66:         (G*(_simplex[last_simplex_index].getH() - Xg_h)) + Xg_h,
67:         (G*(_simplex[last_simplex_index].getK() - Xg_k)) + Xg_k
68:       );
69:       Xc.setMetric(countMetrics(Xc, segment));
70:       // Contraction condition: YES => update the worst parameters; NO =>
    multiple contraction //
71:       if (Xc.getMetric() < _simplex[last_simplex_index].getMetric()){
72:         _simplex[last_simplex_index] = Xc;
73:       }
74:       else {
75:         // -- MULTIPLE CONTRACTION-- //
76:         EquationData tmp_eq;
77:         for (int size = _simplex.size(), k = 1; k < size; k++){ // contraction
    all sets of parameters without the best one
78:           tmp_eq.setAllParameters(
79:             (H*(_simplex[k].getP() - _simplex[0].getP())) + _simplex[0].getP(),
80:             (H*(_simplex[k].getCg() - _simplex[0].getCg())) +
    _simplex[0].getCg(),
81:             (H*(_simplex[k].getC() - _simplex[0].getC())) + _simplex[0].getC(),
82:             (H*(_simplex[k].getDt() - _simplex[0].getDt())) + _simplex[0].getP(),
83:             (H*(_simplex[k].getH() - _simplex[0].getH())) + _simplex[0].getH(),
84:             (H*(_simplex[k].getK() - _simplex[0].getK())) + _simplex[0].getK()
85:           );
86:           _simplex[k] = tmp_eq;
87:         }
88:       }
89:     }
90:     else {
91:       // -- EXPANSION -- //
92:       // Xe //
93:       EquationData Xe(
94:         (B*(Xr.getP() - Xg_p)) + Xr.getP(),
95:         (B*(Xr.getCg() - Xg_cg)) + Xr.getCg(),
96:         (B*(Xr.getC() - Xg_c)) + Xr.getC(),
97:         (B*(Xr.getDt() - Xg_dt)) + Xr.getDt(),
98:         (B*(Xr.getH() - Xg_h)) + Xr.getH(),
99:         (B*(Xr.getK() - Xg_k)) + Xr.getK()
100:       );
101:       Xe.setMetric(countMetrics(Xe, segment));
102:       // Expansion condition: ALWAYS => update the worst parameters; YES =>
    update by Xe; NO => update by Xr //
103:       if (Xe.getMetric() < Xr.getMetric()){
104:         _simplex[last_simplex_index] = Xe;
105:       }
106:       else {
107:         _simplex[last_simplex_index] = Xr;
108:       }
109:     }
110:     sort();
111:   }
112:   // Save the best parameters into segment //
113:   segment.setResult(_simplex[0]);
114: }
```

```
115:
116: void NelderMead::generateSimplex(){
117:   default_random_engine rnd((unsigned)
     chrono::system_clock::now().time_since_epoch().count());
118:
119:   uniform_real_distribution<double> gen_p(EquationBounds::pmin,
     EquationBounds::pmax);
120:   uniform_real_distribution<double> gen_cg(EquationBounds::cgmin,
     EquationBounds::cgmax);
121:   uniform_real_distribution<double> gen_c(EquationBounds::cmin,
     EquationBounds::cmax);
122:   uniform_real_distribution<double> gen_dt(EquationBounds::dtmin,
     EquationBounds::dtmax);
123:   uniform_real_distribution<double> gen_h(EquationBounds::hmin,
     EquationBounds::hmax);
124:   uniform_real_distribution<double> gen_k(EquationBounds::kmin,
     EquationBounds::kmax);
125:
126:   if (!_simplex.empty()){
127:     cout << "Clear simplex before generate new one..." << endl;
128:     _simplex.clear();
129:   }
130:
131:   EquationData eq;
132:   for (int i = 0; i < NUM_OF_PARAM_SETS; i++){
133:     eq.setAllParameters(gen_p(rnd), gen_cg(rnd), gen_c(rnd), gen_dt(rnd),
     gen_h(rnd), gen_k(rnd));
134:     _simplex.push_back(eq);
135:   }
136: }
137:
138: void NelderMead::countAllMetrics(SegmentData &segment){
139:   double metric;
140:   bool least_one_good = false;
141:   for (int size = _simplex.size(), i = 0; i < size; i++){
142:     metric = countMetrics(_simplex[i], segment);
143:     _simplex[i].setMetric(metric);
144:     if (metric != DBL_MAX){
145:       least_one_good = true;
146:     }
147:   }
148:   if (!least_one_good){
149:     cout << "Any metric isnt good => generate new simplex..." << endl;
150:     generateSimplex();
151:     countAllMetrics(segment);
152:   }
153: }
154:
155: double NelderMead::countMetrics(EquationData &params, SegmentData &segment){
156:   double current_blood_time;
157:   double eq_fi, eq_alfa, eq_beta, eq_gama, eq_D, eq_b;
158:   double relative_error, total_relative_error, sum_relative_error = 0, deviation
     = 0, sum_deviation = 0;
159:   int blood_values_count = segment.getBloodValues().size();
160:   int total_count_metrics_values = blood_values_count;
161:   for (int i = 0; i < blood_values_count; i++){
162:     current_blood_time = segment.getBloodValues()[i].getDate();
163:
164:     eq_fi = current_blood_time + params.getDt() + (params.getK() *
     ((segment.getInterpolationIst(current_blood_time) -
     segment.getInterpolationIst(current_blood_time - params.getH())) /
     params.getH()));
165:     if (eq_fi < segment.getMinDate() || eq_fi > segment.getMaxDate()){
166:       total_count_metrics_values--;
```

```
167:       continue;
168:     }
169:     eq_alfa = params.getCg();
170:     eq_beta = params.getP() - (eq_alfa *
     segment.getInterpolationIst(current_blood_time));
171:     eq_gama = params.getC() - segment.getInterpolationIst(eq_fi);
172:     eq_D = pow(eq_beta, 2) - (4 * eq_alfa*eq_gama);
173:     if (eq_D < 0.0) eq_D = 0.0;
174:     eq_b = (-eq_beta + sqrt(eq_D)) / (2 * eq_alfa);
175:
176:     relative_error = abs(segment.getBloodValues()[i].getBlood() - eq_b) /
     segment.getBloodValues()[i].getBlood();
177:     sum_relative_error += relative_error;
178:     sum_deviation += pow(relative_error, 2);
179:   }
180:
181:
182:   if (total_count_metrics_values == 0){
183:     return DBL_MAX;
184:   }
185:   else {
186:     total_relative_error = sum_relative_error / total_count_metrics_values;
187:     deviation = sqrt((sum_deviation / total_count_metrics_values) -
     pow(total_relative_error, 2));
188:     return total_relative_error + deviation;
189:   }
190: }
191:
192: void NelderMead::sort(){
193:   std::sort(_simplex.begin(), _simplex.end());
194: }
```