main.cpp

```
/*! \brief The main function contains the user's constructed code for performing
optimization on the problem.
                        In this particular example, the main function contains the
code for optimizing feedback policy of an adaptive phase estimation scheme.
*/

#include <iostream>

#include "phase_loss_opt.h" //The header file for the specific problem
#include "mpi_optalg.h" //The header file for the optimization algorithms.'mpi.h'
is included in this header.
#include "io.h" //The header file for user-specified parameters

#include <time.h>
#include "corecrt_math_defines.h"

using namespace std;

int main(int argc, char** argv) {

        /*mpi handlers*/
        int my_rank; //processor ID
        int nb_proc; //number of processors
        time_t start_time;
        MPI_Status status;
        int tag = 1;
        int* can_per_proc; //array containing the number of candidate solutions on
each of the processors

        /*variables for the problem*/
        int numvar; //number of variables to be optimized
        double* solution; //array for containing the solution
        double* fitarray; //array containing the fitness values
        int p, t, T = 0;
        double final_fit; //the optimum fitness value
        double* soln_fit; //array for containing the fitness values for all
candidate solutions
        bool mem_ptype[2] = { false, false }; //array for storing the type of
policy -- specific to adaptive phase estimation
        double memory_forT[2]; //array for storing the fitness values for
T_condition

        /*parameter settings*/
        int pop_size, iter, iter_begin, repeat, seed, data_end;
        int N_begin, N_cut, N_end; //variables for setting begin and ending of
numvar
        double prev_dev, new_dev;
        double t_goal; //the acceptable level of error: in this case, it is the
```

```
confidence interval
        string output_filename, time_filename, optimization;
        char const* config_filename;

        /*reading the parameters from io*/
        if (argc > 1) {
                config_filename = argv[1];
        }
        else {
                config_filename = NULL;
        }
        read_config_file(config_filename, &pop_size, &N_begin, &N_cut, &N_end,
&iter,
                &iter_begin, &repeat, &seed, &output_filename,
                &time_filename, &optimization, &prev_dev, &new_dev, &t_goal,
&data_end);

        prev_dev = prev_dev * M_PI;
        new_dev = new_dev * M_PI;

        /*specifying data collection over many numvar*/
        int data_start = N_begin;
        int data_size = data_end - data_start;

        /*start mpi*/
        MPI_Init(&argc, &argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
        MPI_Comm_size(MPI_COMM_WORLD, &nb_proc);

        int num_repeat = 10 * N_end * N_end; //The size of the samples used in the
the learning algorithm
        soln_fit = new double[pop_size]; //an array to store global fitness from
each candidate.
        solution = new double[N_end]; //an array to store solution

        //build memory_fitarray
        int memory_fitarray_size = N_end - data_start + 1;
        double** memory_fitarray = new double* [memory_fitarray_size];
        for (int i = 0; i < memory_fitarray_size; i++)
                memory_fitarray[i] = new double[2];

        //double memory_fitarray[N_end - data_start + 1][2]; //memory for storing
data from many numvar's to be used in accept/reject criteria

        /*Initializing the RNG*/
        //Maximum number of uniform random numbers we will use in one go
        int n_urandom_numbers = num_repeat + 2 * num_repeat * N_end;
        //Maximum number of Gaussian random numbers we will use in one go
        int n_grandom_numbers = 3 * num_repeat * N_end;
        Rng* gaussian_rng = new Rng(true, n_grandom_numbers, seed, my_rank);
        Rng* uniform_rng = new Rng(false, n_urandom_numbers, seed, my_rank);

        //calculating number of candidate per processor and stores the number in
an array.
```

```
        can_per_proc = new int[nb_proc];
        for (p = 0; p < nb_proc; ++p) {
                can_per_proc[p] = 0; //make sure the array started with zeroes.
        }
        for (p = 0; p < pop_size; ++p) {
                can_per_proc[p % nb_proc] += 1;
        }

        if (my_rank == 0) {
                output_header(output_filename.c_str(), time_filename.c_str());
        }

        numvar = N_begin;

        /*beginning the optimization algorithm*/
        if (my_rank == 0)
        {
                cout << endl;
                cout << "A new experiment." << endl;
                cout << "Beginning the optimization algorithms" << endl;
        }

        int j = 0;      // just used for print message

        do {

                t = 0;

                //instantiate the particular problem using pointer from Problem
class
                Problem* problem;
                try {
                        problem = new Phase(numvar, gaussian_rng, uniform_rng);
                }
                catch (invalid_argument) {
                        numvar = 4;
                        problem = new Phase(numvar, gaussian_rng, uniform_rng);
                }
                //instantiating the particular algorithm using pointer from OptAlg
class
                OptAlg* opt;
                if (optimization == "de") {
                        opt = new DE(problem, gaussian_rng, pop_size);
                }
                else if (optimization == "pso") {
                        opt = new PSO(problem, gaussian_rng, pop_size);
                }
                else {
                        throw runtime_error("Unknown optimization algorithm");
                }

                fitarray = new double[problem->num_fit];

                /*start timer*/
                start_time = time(NULL);

                //Initializa population
                if (numvar < N_cut) {
                        //If the user chose to initialize population uniformly
over the search space
                        try {
                                opt->Init_population(can_per_proc[my_rank]);
                        }
                        catch (invalid_argument) {
                                //cout << "Population size at processor" <<
my_rank << "is <=0." << endl;
                                terminate();
                        }
                }
                else {
                        //If the user chose to initialize population using a
Guassian distribution over a point in the search space.
                        if (my_rank == 0) {
                                for (p = 1; p < nb_proc; ++p) {
                                        MPI_Send(&solution[0], numvar, MPI_DOUBLE,
p, tag, MPI_COMM_WORLD);
                                }
                        }
                        else {
                                MPI_Recv(&solution[0], numvar, MPI_DOUBLE, 0, tag,
MPI_COMM_WORLD, &status);
                        }
                        try {
                                opt->Init_previous(prev_dev, new_dev,
can_per_proc[my_rank], solution);
                                //each processor initialize the candidates.
                        }
                        catch (invalid_argument) {
                                cout << "Population size at processor" << my_rank
<< "is <=0." << endl;
                                terminate();
                        }
                }

                //Copy the candidates that are initialized to personal best
                opt->put_to_best();

                //setting the success criterion
                if (numvar < N_cut) {
                        //The optimization terminate after T step
                        try {
                                opt->set_success(iter_begin, 0);
                        }
                        catch (out_of_range) {
                                iter_begin = 100;
                                opt->set_success(iter_begin, 0);
                        }
                        T = iter_begin;
                }
                else if (numvar >= data_end) {
                        //The optimization terminate after the set level of error
is reached
                        opt->set_success(1000, 1);  //stop after perform 1000
iteration or exceeding the goal.
                }
                else {
                        //The optimization terminate after T step
                        try {
                                opt->set_success(iter, 0);
                        }
                        catch (out_of_range) {
                                iter = 100;
                                opt->set_success(iter, 0);
                        }
                        T = iter;
                }

                /*Start iterative optimization*/
                if (my_rank == 0)
                        cout << "Start iterative optimization.  " << "numvar: " <<
numvar << endl;
                do {
                        ++t;
                        if (my_rank == 0)
                        {
                                if (t % 100 == 0)
                                        cout << "Iterating...   t: " << t << endl;
                        }
                        opt->update_popfit(); //recalculated the fitness values
and update the mean of the fitness values for the population
                        opt->combination(); //create contenders(offsprings). This
is where a lot of comm goes on between processors (zz don't think so)
                        opt->selection(); //select candidates or contenders for
the next step

                        final_fit = opt->Final_select(soln_fit, solution,
fitarray); //communicate to find the best solution that exist so far



                        //check if optimization is successful. This function
includes accept-reject criteria.
                        opt->success = opt->check_success(t, fitarray,
&memory_fitarray[0][0], data_size, t_goal, mem_ptype, &numvar, N_cut,
memory_forT);
                } while (opt->success == 0);

                if (my_rank == 0)
                        cout << "Iteration done" << endl;
                //repeat calculation of fitness value to find the best one in the
population
                final_fit = opt->avg_Final_select(solution, repeat, soln_fit,
fitarray);
```

```
                // Print results after every iterative optimization
                if (my_rank == 0)
                {
                        if (numvar > 3)
                        {
                                // print numvar, solution, fitarray
                                cout << "The result of optimization: " << endl;
                                cout << "numvar: " << numvar << endl;
                                cout << "solution:  ";
                                for (j = 0; j < numvar; j++)
                                {
                                        cout << solution[j] << '\t';
                                }
                                cout << endl;
                                cout << "fitarray:  ";
                                for (j = 0; j < problem->num_fit; j++)
                                {
                                        cout << fitarray[j] << '\t';
                                }
                                cout << endl;
                        }
                }

                // save results to the test file
                if (my_rank == 0)
                {

                }

                // save results to the final file
                if (my_rank == 0) {
                        //if the policy is of the correct type output the solution
                        if ((numvar >= N_begin) && (!mem_ptype[0] &&
!mem_ptype[1])) {
                                //              if ((numvar >= N_begin)) {
                                output_result(numvar, problem->num_fit, fitarray,
solution, start_time,
                                        output_filename.c_str(),
time_filename.c_str());
                        }
                }

                //collect data to use in error calculation
                if (numvar >= data_start && numvar < data_end) {
                        memory_fitarray[numvar - data_start][0] = log10(numvar);
                        memory_fitarray[numvar - data_start][1] =
log10(pow(final_fit, -2) - 1);
                }
                else if (numvar >= data_end) {
                        ++data_size;
                }
                else {}
                ////testing how the solution perform under lossy condition
                //if (my_rank == 0) {
                //      problem->fitness(solution, fitarray);
                //      //final_fit = fitarray[0];
                //      cout << numvar << "\t" << fitarray[0] << "\t" <<
fitarray[1] << endl;
                //}
                if (my_rank == 0)
                        cout << endl << endl;

                /*delete the objects for algorithm and problem to free memory*/
                delete opt;
                delete problem;

                ++numvar;

        } while (numvar <= N_end);

        delete gaussian_rng;
        delete uniform_rng;
        delete[] solution;
        delete[] soln_fit;
        delete[] can_per_proc;
        for (int i = 0; i < memory_fitarray_size; i++)
                delete[] memory_fitarray[i];
        delete[] memory_fitarray;

        MPI_Finalize();
        if (my_rank == 0) {
                cout << "done" << endl;
        }

        //pause before exit
        cin.get();

        return 0;
}
```

aux_functions.cpp

```
#include <iostream>
#include "aux_functions.h"

#include "corecrt_math_defines.h"

using namespace std;

/*############################Linear Regression############################*/
void linear_fit(int data_size, double *x, double *y, double *slope, double
*intercept, double *mean_x) {
    /** This function calculates the coefficients of a linear equation.
    * The function takes two arrays of x and y of size data_size, and output the
slope, intercept, and the mean_x.
    */
    if(data_size <= 0) {
        throw invalid_argument("data_size must be positive.");
        }
    double sum_x = 0;
    double sum_xx = 0;
    double sum_y = 0;
    double sum_xy = 0;

    //Preparing the data for calculating mean_x, slope, and intercept.
    for(int v = 0; v < data_size; ++v) {
        sum_x = sum_x + x[v];
        sum_xx = sum_xx + x[v] * x[v];
        sum_y = sum_y + y[v];
        sum_xy = sum_xy + x[v] * y[v];
        }

    *mean_x = sum_x / double(data_size);
    *slope = (sum_xy - sum_y * sum_x / double(data_size)) / (sum_xx - sum_x *
sum_x / double(data_size));
    *intercept = sum_y / double(data_size) - *slope * (*mean_x);
    }

double error_interval(double *x, double *y, double mean_x, int data_size, double
slope, double intercept) {
    /** This function calculates the error of the latest data y in the array.
    * This requires all data x and y of size data_size, slope, mean_x, and
intercept from linear_fit function.
    * SSres is a variable used to store the value of (x-mean_x)^2, which can be
used to reduce the time for calculating error_update.
    */
    if(data_size <= 0) {
        throw invalid_argument("data_size must be positive.");
        }
    double SSx = 0;
    double SSres = 0;

    for(int i = 0; i < data_size; ++i) {
        SSres = SSres + pow(y[i] - slope * x[i] - intercept, 2);
        SSx = SSx + (x[i] - mean_x) * (x[i] - mean_x);
        }
    return sqrt(SSres / double(data_size - 1) * (1 / data_size + (pow(x[data_size]
- mean_x, 2) / SSx)));
    }

double error_update(int old_size, double *SSres, double *mean_x, double slope,
double intercept, double *y, double *x) {
    /** This function update the error_interval when a new value of y.
    * The function requires the coefficients of the linear equations and the data
array x and y.
    */
    if(old_size <= 0) {
        throw invalid_argument("data_size must be positive.");
        }
    double SSx = 0;

    *mean_x = (*mean_x * old_size + x[old_size]) / double(old_size + 1);
//updating mean_x
    *SSres = *SSres + pow(y[old_size] - slope * x[old_size] - intercept, 2);

    for(int i = 0; i < old_size + 1; ++i) {
        SSx = SSx + (x[i] - *mean_x) * (x[i] - *mean_x);
        }
    return sqrt(*SSres / double(old_size - 1) * (1 / old_size + (pow(x[old_size] -
*mean_x, 2) / SSx)));
    }

/*###########################Calculating Quantile#############################*/
double quantile(double p) { //p is percentile
    double result;
    try {
        result = inv_erf(p);
        }
    catch(invalid_argument) {
        p = 0.999999;
        }
    return sqrt(2) * inv_erf(2 * p - 1);
    }

inline double inv_erf(double x) {
    if(x == 1) {
        throw invalid_argument("Input leads to error.");
        }
    double a = 0.140012;
    double lnx = log(1 - x * x);
    double temp = sqrt(pow(2.0 / (M_PI * a) + lnx / 2.0, 2) - lnx / a);

    return sgn(x) * sqrt(temp - 2.0 / (M_PI * a) - lnx / 2.0);
    }
```

```
inline int sgn(double x) {
    /** Sign function of x: https://en.wikipedia.org/wiki/Sign_function
    */
    if(x < 0) {
        return -1;
        }
    else if(x == 0) {
        return 0;
        }
    else {
        return 1;
        }
    }
```

aux_functions.h

```
#include <stdexcept>
#include <cmath>

using namespace std;

/*! \brief This library contains the functions for linear regression.
*/

//calculating linear regression
void linear_fit(int data_size, double *x, double *y, double *slope, double
*intercept, double *mean_x); /*!< Function for calculating the linear regression
from a set of x,y array of size data_size.*/
double error_interval(double *x, double *y, double mean_x, int data_size, double
slope, double intercept);/*!< Function for calculating the error of the current y
data using the previous x and y data and the linear equation calculated from
linear_fit.*/
double error_update(int old_size, double *SSres, double *mean_x, double slope,
double intercept, double *y, double *x); /*!< Function for updating the error
interval for new value of y data. This function has to be used after
error_interval.*/
//calculating quantile
double quantile(double p); /*!< Function for calculating the quantile of a normal
distribution corresponding to the percentile p.*/
inline double inv_erf(double x); /*!< Function for calculating the inverse of an
error function.*/
inline int sgn(double x); /*!< Sign function.*/
```

candidate.cpp

```
#include "candidate.h"

Candidate::~Candidate() {
    if (is_candidate_initialized) {
        delete[] can_best;
        delete[] contender;
        delete[] global_best;

        delete[] best_fit;
        delete[] cont_fit;
        delete[] global_fit;
        }

    if (is_velocity_initialized) {
        delete[] velocity;
        }
    }

void Candidate::init_can(int numvar, /*!<number of variables*/
                         int fit_size /*!< the number of objective functions*/
                        ) {
    if(numvar <= 0) {
        throw out_of_range("numvar must be positive.");
        }
    if(fit_size <= 0) {
        throw invalid_argument("num_fit must be positive.");
        }
    num = numvar;
    can_best = new double[num];
    contender = new double[num];
    global_best = new double[num];

    num_fit = fit_size;
    best_fit = new double[num_fit];
    cont_fit = new double[num_fit];
    global_fit = new double[num_fit];
    is_candidate_initialized = true;
    }

void Candidate::init_velocity() {
    velocity = new double[num];
    is_velocity_initialized = true;
    }

void Candidate::update_cont(double *input) {
    memcpy(contender, input, num * sizeof(double));
    }
```

```
void Candidate::update_vel(double *input) {
    memcpy(velocity, input, num * sizeof(double));
    }

void Candidate::update_best() {
    double *temp_pnt;

    temp_pnt = can_best;
    can_best = contender;
    contender = temp_pnt;

    temp_pnt = best_fit;
    best_fit = cont_fit;
    cont_fit = temp_pnt;

    best_times = times;
    }

void Candidate::update_global(double *input) {
    memcpy(global_best, input, num * sizeof(double));
    }

void Candidate::put_to_global() {
    //swapping pointer does not work here
    memcpy(global_best, can_best, num * sizeof(double));
    memcpy(global_fit, best_fit, num_fit * sizeof(double));
    global_times = best_times;
    }

void Candidate::read_cont(double *output) {
    memcpy(output, contender, num * sizeof(double));
    }

void Candidate::read_vel(double *output) {
    memcpy(output, velocity, num * sizeof(double));
    }

void Candidate::read_best(double *output) {
    memcpy(output, can_best, num * sizeof(double));
    }

void Candidate::read_global(double *output) {
    memcpy(output, global_best, num * sizeof(double));
    }

void Candidate::write_contfit(double *fit, int tt) {
    memcpy(cont_fit, fit, num_fit * sizeof(double));
    times = tt;
    }

void Candidate::write_bestfit(double *fit) {
    for(int i = 0; i < num_fit; i++) {
        best_fit[i] = best_fit[i] * double(best_times) + fit[i];
        }
    best_times += 1;
    for(int i = 0; i < num_fit; i++) {
        best_fit[i] = best_fit[i] / double(best_times);
        }
    }

void Candidate::write_globalfit(double *fit) {
    memcpy(global_fit, fit, num_fit * sizeof(double));
    }
```

candidate.h

```
#ifndef CANDIDATE_H
#define CANDIDATE_H
#include <stdexcept>
#include <cstring>

using namespace std;

/*! \brief Candidate class contains arrays to be used by optimization algorithm to
store data of a solution candidate.
*/

class Candidate {

        friend class OptAlg;

        /** OptAlg class is friend of Candidate class and can access the arrays of
solution candidates directly.
                *   However, the specific algorithms are not.
                */

    public:
        Candidate() {
            is_velocity_initialized = false;
            is_candidate_initialized = false;
            };
        ~Candidate();

        void init_can(int numvar, int fit_size);/*!< A function for allocating
memories for storing information for a candidate solution.*/
        void init_velocity();/*!< A function for allocating memory for additional
information: velocity.*/

        void update_cont(double *input); /*!< A function for copying onto the
contender array from input.*/
        void update_vel(double *input); /*!< A function for copying onto the
velocity array from input.*/
        void update_best(); /*!< A function for putting the contender array to the
can_best array.This is done by swapping pointers.*/
        void update_global(double *input);  /*!< A function for copying onto the
global_best array from input.*/
        void put_to_global(); /*!< A function for copying can_best array to
global_best array.*/

        void read_cont(double *output); /*!< A function for reading the contender
array to output array.*/
        void read_vel(double *output); /*!< A function for reading the velocity
array to output array.*/
        void read_best(double *output); /*!< A function for reading the can_best
```

```
array to output array.*/
        void read_global(double *output); /*!< A function for reading the
global_best array to output array.*/

        void write_contfit(double *fit, int tt); /*!< A function for copying an
array of fitness values to cont_fit array.*/
        void write_bestfit(double *fit); /*!< A function for copying an array of
fitness values to best_fit array.*/
        void write_globalfit(double *fit); /*!< A function for copying an array of
fitness values to global_fit array.*/

        double read_contfit(int i /*!< index of fitness value to be read*/) { /*!
A function for reading a fitness value.*/
            return cont_fit[i];
            }
        double read_bestfit(int i /*!< index of fitness value to be read*/) {/*! A
function for reading a fitness value.*/
            return best_fit[i];
            }
        double read_globalfit(int i /*!< index of fitness value to be read*/) {/*!
A function for reading a fitness value.*/
            return global_fit[i];
            }
        int read_bestt() {/*! A function for reading the number of times the
best_fit has been averaged over.*/
            return best_times;
            }


    private:

        int num; /*number of variables*/
        int num_fit; /*number of fitness values (i.e., total number of contraints
and objetive functions)*/
        double *best_fit; /*pointer to array for storing fitness values calculated
from the data in contender array*/
        double *cont_fit; /*pointer to array for storing fitness values calculated
from the data in can_best array*/
        double *global_fit; /*pointer to array for storing fitness values
calculated from the data in global_best array*/
        int times; /*the number of time the fitess value is computed for
contender*/
        int best_times; /*the number of time the fitess value is computed for
can_best*/
        int global_times; /*the number of time the fitess value is computed for
global_best*/
        bool is_velocity_initialized; /*indicating whether addition memory needs
to be allocated*/
        bool is_candidate_initialized; /*indicating whether memories are
allocated*/

        //memory arrays
        double *can_best; /*the pointer to array that contains solution that would
survive several iterations*/
```

```
        double *contender; /*the pointer to array that serves mostly as temporary
memory that does not survive an interation*/
        double *velocity; /*the pointer to additional array*/
        double *global_best; /*the pointer to array that contains solution to be
compared for the final solution*/
    };

#endif // CANDIDATE_H
```

io.cpp

```
#include <fstream>
#include <iostream>
#include <sstream>
#include <string>
#include <stdexcept>
#include <map>
#include <time.h>

using namespace std;

void output_header( char const *output_filename, /*!<Pointer to output file where
the sharpness will be printed.*/
                    char const *time_filename /*!<Pointer to output file where the
clock time for optimizing a policy.*/
                  ) {
    ofstream output_file;
    ofstream time_file;

    output_file.open(output_filename, ios::app);
    output_file << "#N \t Sharpness \t Mean \t Policy" << endl;
    output_file.close();
    time_file.open(time_filename, ios::app);
    time_file << "#N \t Time" << endl;
    time_file.close();
    }

void output_result( int num, /*!< Number of variables in a policy.*/
                    int num_fit, /*!<number of fitness values*/
                    double *final_fit, /*!<fitness values.*/
                    double *solution, /*!< Policy*/
                    time_t start_time, /*!< Clock time for finding the policy.*/
                    char const *output_filename, /*!< Pointer to file that store
the fitness value and the policy.*/
                    char const *time_filename /*!< Pointer to file that store
time.s*/
                  ) {
    int i;
    ofstream output_file;
    ofstream time_file;

    output_file.open(output_filename, ios::app);
    output_file << num << "\t";
    for(i = 0; i< num_fit; i++){
        output_file << final_fit[i] << "\t";
    }
    for(i = 0; i < num; ++i) {
        output_file << solution[i] << "\t";
        }
    output_file << endl;
    output_file.close();

    time_file.open(time_filename, ios::app);
    time_file << num << "\t" << difftime(time(NULL), start_time) << endl;
    time_file.close();

    }

void output_run_data(const char* str, char const* run_data_file_name)
{
        ofstream run_data_file;
        run_data_file.open(run_data_file_name, ios::app);
        run_data_file << str;
        run_data_file.close();
        cout << str;
}

//void output_result(int num, double final_fit, double *solution,
//                   time_t start_time,
//                   char const *output_filename,
//                   char const *time_filename);

map<string, string> parse(ifstream & cfgfile) {
    map<string, string> options;
    string id, eq, val;

    while(cfgfile >> id >> eq >> val) {
        if (id[0] == '#') continue;  // skip comments
        //cout << id << " " << eq << " " << val << endl;
        if (eq != "=") throw runtime_error("Parse error");

        options[id] = val;
        }
    return options;
    }

void read_config_file(  char const *filename, /*!< File that contains program
parameters.*/
                        int *pop_size, int *N_begin,
                        int *N_cut, int *N_end, int *iter, int *iter_begin,
                        int *repeat, int *seed, string *output_filename,
                        string *time_filename, string *optimization,
                        double *prev_dev, double *new_dev, double *t_goal, int
*data_end) {
    /*! This fucntion reads the user specified configuration file or set the
parameters to defaults.
    Parameters that can be set by users are
    population size: *pop_size
    smallest number of variables to be searched: *N_begin
    number of variables to switch from uniform initialization: *N_cut
    largest number of variables: *N_end
    number of iteration when initialization is uniformed: *iter_begin
    number of iteration before accepting/checking a policy: *iter
    number of times the fitness value is compute in the final selection process:
*repeat
    seed for rng: *seed
    the name of the file that the fitness value and policy will be stored:
*output_filename
    the name of the file that the clock time will be stored: *time_filename =
"time.dat"
    name of the optimization algorithm: *optimization = "de"
    parameter for initialization using previous result -- for variables with
previous data: *prev_dev = 0.01;
    parameter for initialization using previous result -- for new variables:
*new_dev = 0.25;
    goal to be used in accept/reject criterion *t_goal
    the number of variable to begin using accept/reject criterion: *data_end
    */
    // Setting defaults

        // Origin setting
    //*pop_size = 12;
    //*N_begin = 4;
    //*N_cut = 5;
    //*N_end = 10;
    //*iter = 100;
    //*iter_begin = 300;
    //*repeat = 10;
    //*seed = time(NULL);
    //*output_filename = "output.dat";
    //*time_filename = "time.dat";
    //*optimization = "pso";

    //*data_end = 8;
    //*prev_dev = 0.01;
    //*new_dev = 0.25;
    //*t_goal = 0.98; //probability for calculating quantile

        // zz test setting
        *pop_size = 12;
        *N_begin = 4;
        *N_cut = 5;
        *N_end = 10;
        *iter = 200;
        *iter_begin = 300;
        *repeat = 10;
        *seed = time(NULL);
        *output_filename = "output.dat";
        *time_filename = "time.dat";
        *optimization = "pso";

        *data_end = 8;
        *prev_dev = 0.01;
        *new_dev = 0.25;
        *t_goal = 0.98; //probability for calculating quantile

    // Parsing config file if there is one
    if (filename == NULL) return;
    ifstream config_file(filename);
    if(!config_file.is_open()) throw runtime_error("Config file cannot be
opened!");
    map<string, string> options = parse(config_file);
    config_file.close();
    map<string, string>::iterator it;
    for (it = options.begin(); it != options.end(); ++it) {
        if (it->first == "pop_size") {
            istringstream (it->second) >> *pop_size;
            }
        else if (it->first == "N_begin") {
            istringstream (it->second) >> *N_begin;
            }
        else if (it->first == "N_cut") {
            istringstream (it->second) >> *N_cut;
            }
        else if (it->first == "N_end") {
            istringstream (it->second) >> *N_end;
            }
        else if (it->first == "iter") {
            istringstream (it->second) >> *iter;
            }
        else if (it->first == "iter_begin") {
            istringstream (it->second) >> *iter_begin;
            }
        else if (it->first == "repeat") {
            istringstream (it->second) >> *repeat;
            }
        else if (it->first == "output_filename") {
            *output_filename = it->second;
            }
        else if (it->first == "time_filename") {
            *time_filename = it->second;
            }
        else if (it->first == "random_seed") {
            istringstream (it->second) >> *seed;
            }
        else if (it->first == "optimization") {
            *optimization = it->second;
            if (*optimization != "de" && *optimization != "pso") {
                throw runtime_error("Unknown optimization algorithm");
                }
            }
        else if (it->first == "data_end") {
            istringstream (it->second) >> *data_end;
            }
        else if (it->first == "prev_dev") {
            istringstream (it->second) >> *prev_dev;
            }
        else if (it->first == "new_dev") {
            istringstream (it->second) >> *new_dev;
            }
        else if (it->first == "t_goal") {
            istringstream (it->second) >> *t_goal;
            }
        else {
            throw runtime_error("Unknown configuration option");
            }
        }
    if (*N_cut < *N_begin) {
        throw runtime_error("Select new N_cut");
        }
    }
```

io.h

```
#ifndef IO_H
#define IO_H

/*! \brief This file contains the functions that are involved in setting the input
parameters
 and printing out the results.
*/

void output_header(char const *output_filename, char const *time_filename); /*!
<This function print the headers to output files.*/

void output_result(int num, int num_fit, double *final_fit, double *solution,
                   time_t start_time, char const *output_filename,
                   char const *time_filename); /*!<This function print the
sharpness, policy, and time for N.*/

void output_run_data(const char* str, char const* run_data_dile_name); // output
run data

void read_config_file(char const *filename, int *pop_size, int *N_begin,
                      int *N_cut, int *N_end, int *iter, int *iter_begin,
                      int *repeat, int *seed, string *output_filename,
                      string *time_filename, string *optimization,
                      double *prev_dev, double *new_dev, double *t_goal, int
*data_end);/*!<This function set the parameters for the optimization algorithm.*/

#endif // IO_H
```

mpi_optalg.cpp

```
#include "mpi_optalg.h"
#include <iostream>

/*###############################Function for initializing
population###############################*/

void OptAlg::Init_population(int psize)
{
        /*! This function first initialize a group of candidates for a processor
uniformly over the search space.
        * The function then compute the fitness values for each.
        */
        if (psize <= 0)
                throw invalid_argument("Population size must be positive.");

        double* input = new double[num];

        //store the variables
        pop_size = psize;
        pop = new Candidate[pop_size];

        for (int p = 0; p < pop_size; ++p)
        {
                //generate candidate uniformly
                for (int i = 0; i < num; ++i)
                        input[i] = double(rand()) / RAND_MAX * (prob-
>upper_bound[i] - prob->lower_bound[i]) + prob->lower_bound[i];


                //store it in candidate object
                pop[p].init_can(num, num_fit); //First the memories in the object
are initialized.
                pop[p].update_cont(input); //The candidates are stored in the
contender array.
                Cont_fitness(p); //The fitness values are computed.
        }

        delete[] input;
}

void OptAlg::Init_previous(double prev_dev, double new_dev, int psize, double*
prev_soln) {
        /*! This function first initialize a group of candidates for a processor
from a previous solution.
        * The function then compute the fitness values for each.
        */
        if (psize <= 0) {
                throw invalid_argument("Population size must be positive");
```

```
	}
	int dev_cut_off = num - 1;

	double* input = new double[num];
	double* dev = new double[num];

	//store the variables
	pop_size = psize;
	pop = new Candidate[pop_size];

	prev_soln[num - 1] = prev_soln[num - 2]; //Generate the input array from
previous solution.
	dev_gen(dev, prev_dev, new_dev, dev_cut_off); //Generate an array of the
width of the Gaussian distribution.

	for (int p = 0; p < pop_size; ++p) {
		//generate candidate
		for (int i = 0; i < num; ++i) {
			input[i] = fabs(gaussian_rng->next_rand(prev_soln[i],
dev[i]));
		}
		prob->boundary(input);
		//store it in candidate object
		pop[p].init_can(num, num_fit); //First the memories in the object
are initialized.
		pop[p].update_cont(input); //The candidates are stored in the
contender array.
		Cont_fitness(p); //The fitness values are computed.
	}

	delete[] input;
	delete[] dev;
}

/*################################Function for calculating
fitnesses################################*/

void OptAlg::Cont_fitness(int p)
{
	/*! The mean fitness value of a contender are computed for 2 samples.
	The number of samples is currently fixed.
	*/

	double* fit1 = new double[num_fit];
	double* fit2 = new double[num_fit];

	prob->avg_fitness(pop[p].contender, prob->num_repeat, fit1);
	prob->avg_fitness(pop[p].contender, prob->num_repeat, fit2);
	for (int i = 0; i < num_fit; i++) {
		fit1[i] += fit2[i];
		fit1[i] = fit1[i] / 2.0;
	}
	pop[p].write_contfit(fit1, 2);
```

```
        delete[] fit1;
        delete[] fit2;
}

void OptAlg::Best_fitness(int p)
{
        /*! A fitness value of the best array is computed, and the mean fitness
value is update.
        */

        double* fit = new double[num_fit];

        // calculate the fitness value of a solution
        prob->avg_fitness(pop[p].can_best, prob->num_repeat, fit);
        // this method can calculate and update the mean of best fitness values in
real time.
        pop[p].write_bestfit(fit);

        delete[] fit;
}

void OptAlg::update_popfit() {
        /*! The mean fitness values of the population on a processor are updated.
        */
        // ????? Does the pop_size is the can_per_proc or the total pop_size?
?????//
        // ????? Tend to check the MPI variances ?????//
        for (int p = 0; p < pop_size; ++p) {
                Best_fitness(p);
        }
}

/*##############################Final
Selections##################################*/
double OptAlg::Final_select(double* fit, double* solution, double* fitarray) {
        /*! The fitness values are sent to the zeroth processor, where the
candidate with the highest fitness value.
        * A copy of that candidate to other processors.
        */
        int indx;       // The index of candidate
        MPI_Status status;
        int tag = 1;
        double global_fit;

        fit_to_global();//ensuring that global_best contains the solutions

        //Send all fitness value to root (p=0)

        for (int p = 0; p < total_pop; ++p) {
                if (p % nb_proc == 0) {// if p is on root, then just write the
fitness to central table
                        if (my_rank == 0) {
                                fit[p] = pop[int(p / nb_proc)].read_globalfit(0);
                        }
                }
                else { //if p is not on the root, then read the fitness and send
it to central table in root
                        if (my_rank == p % nb_proc) { //send the fitness from non-
root
                                global_fit = pop[int(p /
nb_proc)].read_globalfit(0);
                                MPI_Send(&global_fit, 1, MPI_DOUBLE, 0, tag,
MPI_COMM_WORLD);
                        }
                        else if (my_rank == 0) { //root receive and store fitness
value
                                MPI_Recv(&fit[p], 1, MPI_DOUBLE, p % nb_proc, tag,
MPI_COMM_WORLD, &status);
                        }
                        else {}
                }
        }

        MPI_Barrier(MPI_COMM_WORLD);

        //find the candidate that with highest fitness and send the index to all
processors.
        if (my_rank == 0) {
                indx = find_max(fit);
                //root send the index to all processors.
                for (int p = 1; p < nb_proc; ++p) {
                        MPI_Send(&indx, 1, MPI_INT, p, tag, MPI_COMM_WORLD);
                }
        }
        //processors other than root receive the index
        else if (my_rank != 0) {
                MPI_Recv(&indx, 1, MPI_INT, 0, tag, MPI_COMM_WORLD, &status);
        }
        else {}

        MPI_Barrier(MPI_COMM_WORLD);

        //read the fitarray and send it to all processor for checking success
criteria
        if (my_rank == indx % nb_proc) {

                for (int i = 0; i < num_fit; ++i) {
                        fitarray[i] = pop[indx / nb_proc].read_globalfit(i);
                }

                for (int p = 0; p < nb_proc; ++p) {
                        if (p != my_rank) {
                                MPI_Send(&fitarray[0], num_fit, MPI_DOUBLE, p,
tag, MPI_COMM_WORLD);
                        }
                }
        }
        else if (my_rank != indx % nb_proc) {
                MPI_Recv(&fitarray[0], num_fit, MPI_DOUBLE, indx % nb_proc, tag,
MPI_COMM_WORLD, &status);
        }
        else {}

        MPI_Barrier(MPI_COMM_WORLD);

        //sending the solution back to root //need to check data type
        if (indx % nb_proc == 0) { //if solution is in root, then read it out.
                if (my_rank == 0) {
                        pop[int(indx / nb_proc)].read_global(solution);
                }
                else {}
        }
        else { //if solution is not in root, send to root
                if (my_rank == indx % nb_proc) {
                        pop[int(indx / nb_proc)].read_global(solution);
                        MPI_Send(&solution[0], num, MPI_DOUBLE, 0, tag,
MPI_COMM_WORLD);
                }
                else if (my_rank == 0) {
                        MPI_Recv(&solution[0], num, MPI_DOUBLE, indx % nb_proc,
tag, MPI_COMM_WORLD, &status);
                }
                else {}
        }

        MPI_Barrier(MPI_COMM_WORLD);

        return fitarray[0];
}

double OptAlg::avg_Final_select(double* solution, int repeat, double* soln_fit,
double* fitarray) {
        /*! Similar to Final_select(), the function finds the solution with the
highest fitness in the population.
        * But before doing so, the fitness values were computed for 10 times (the
variable is repeat).
        */
        MPI_Status status;
        int tag = 1;
        double final_fit;
        int indx;

        double* array = new double[num];
        double** fit = new double* [pop_size];
        for (int i = 0; i < pop_size; i++)
                fit[i] = new double[num_fit];

        //double fitarray[num_fit];

        fit_to_global();//move solution to global_best array in case we're using
DE.
```

```
        //Need to calculate fitness again for 'repeat' times, independently on
each
        for (int p = 0; p < pop_size; ++p) {
                //pop[p].read_global(array);
                fit[p][0] = 0;
                fit[p][1] = 0;
                for (int i = 0; i < repeat; ++i) {
                        prob->avg_fitness(pop[p].global_best, prob->num_repeat,
fitarray);
                        fit[p][0] += fitarray[0];
                        fit[p][1] += fitarray[1];
                }
                fit[p][0] = fit[p][0] / repeat;
                fit[p][1] = fit[p][1] / repeat;
        }

        //filling the fitness table in root
        for (int p = 0; p < total_pop; ++p) {
                if (p % nb_proc == 0) {
                        soln_fit[p] = fit[p / nb_proc][0]; //no need for
transmitting data
                }
                else {
                        if (my_rank == p % nb_proc) {
                                MPI_Send(&fit[p / nb_proc][0], 1, MPI_DOUBLE, 0,
tag, MPI_COMM_WORLD);
                        }
                        else if (my_rank == 0) {
                                MPI_Recv(&soln_fit[p], 1, MPI_DOUBLE, p % nb_proc,
tag, MPI_COMM_WORLD, &status);
                        }
                        else {}
                }
        }

        MPI_Barrier(MPI_COMM_WORLD);

        // find the maximum fitness and send out the fitness for success criteria
testing
        if (my_rank == 0) {
                indx = find_max(soln_fit);
                final_fit = soln_fit[indx];
                for (int p = 1; p < nb_proc; ++p) {
                        MPI_Send(&final_fit, 1, MPI_DOUBLE, p, tag,
MPI_COMM_WORLD);
                }
        }
        else {
                MPI_Recv(&final_fit, 1, MPI_DOUBLE, 0, tag, MPI_COMM_WORLD,
&status);
        }

        MPI_Barrier(MPI_COMM_WORLD);
        // send index of solution to processors
        if (my_rank == 0) {
                for (int p = 1; p < nb_proc; ++p) {
                        MPI_Send(&indx, 1, MPI_INT, p, tag, MPI_COMM_WORLD);
                }
        }
        else {
                MPI_Recv(&indx, 1, MPI_INT, 0, tag, MPI_COMM_WORLD, &status);
        }

        MPI_Barrier(MPI_COMM_WORLD);

        //get solution from the processor
        if (my_rank == indx % nb_proc) {
                //pop[indx / nb_proc].read_global(array);
                // if indx is in root, don't need to send.
                if (my_rank == 0)
                {
                        for (int i = 0; i < num; i++)
                        {
                                solution[i] = pop[indx / nb_proc].global_best[i];
                        }
                }
                // if indx is in other proccessor, send soln to root.
                else
                        MPI_Send(pop[indx / nb_proc].global_best, num, MPI_DOUBLE,
0, tag, MPI_COMM_WORLD);

        }
        else if (my_rank == 0) {
                MPI_Recv(&solution[0], num, MPI_DOUBLE, indx % nb_proc, tag,
MPI_COMM_WORLD, &status);
        }
        else {}

        MPI_Barrier(MPI_COMM_WORLD);

        //get fitarray from the processor
        if (my_rank == indx % nb_proc) {
                // if indx in root, don't need to send.
                if (indx % nb_proc == 0)
                {
                        fitarray[0] = final_fit;
                        fitarray[1] = fit[indx / nb_proc][1];
                }
                // if indx is in other proccessor, send fitness to root.
                else
                {
                        MPI_Send(&fit[indx / nb_proc][1], 1, MPI_DOUBLE, 0, tag,
MPI_COMM_WORLD);
                }
        }
        else if (my_rank == 0) {
                fitarray[0] = final_fit;
                MPI_Recv(&fitarray[1], 1, MPI_DOUBLE, indx % nb_proc, tag,
MPI_COMM_WORLD, &status);
        }
        else {}

        delete[] array;
        for (int i = 0; i < pop_size; i++)
                delete[] fit[i];
        delete[] fit;

        return final_fit;
}


int OptAlg::find_max(double* fit) {
        double* soln;
        int indx;

        soln = &fit[0];
        indx = 0;
        for (int p = 1; p < total_pop; ++p) {

                if (*soln < fit[p]) {
                        soln = &fit[p];
                        indx = p;
                }
                else {}
        }
        return indx;
}

/*#############################Success
Criteria##################################*/

void OptAlg::set_success(int iter, bool goal_in) {
        if (iter <= 0) {
                throw out_of_range("Number of iteration has to be positive.");
        }
        success = 0;
        T = iter;
        goal = goal_in;
}

bool OptAlg::check_success(int t, double* current_fitarray, double*
memory_fitarray, int data_size, double t_goal, bool* mem_ptype, int* numvar, int
N_cut, double* memory_forT) {
        /*! The condition for stopping the optimization is checked. If the
condition is based on the number of iterations,
        * T_condition() is checked. This condition doesn't change t if the
condition is not met, but it reverses numvar
        * so that the optimization process restarts from the initialization. This
function can take the memory from several numvar.
        * If the condition is based on whether the fitness value met the error
condition, error_condition() is called.
```

```
        * This function takes the data from several numvar to generate the error
condition according to t_goal.
        */
        bool type;

        if (goal == 0) {

                if (t >= T) {
                        return prob->T_condition(current_fitarray, numvar, N_cut,
mem_ptype, memory_forT); //This is wrong
                }
                else {
                        return 0;
                }
        }
        else {
                return prob->error_condition(current_fitarray, memory_fitarray,
data_size, t_goal);
        }

}


void OptAlg::dev_gen(double* dev_array, double prev_dev, double new_dev, int
cut_off) {
        for (int i = 0; i < num; ++i) {
                if (i < cut_off) {
                        dev_array[i] = prev_dev;
                }
                else {
                        dev_array[i] = new_dev;
                }
        }
}
```

mpi_optalg.h

```
#ifndef MPIOPTALG_H
#define MPIOPTALG_H
#include <mpi.h>
#include <cstdlib>
#include <stdexcept>
#include <cmath>

#include "problem.h"
#include "candidate.h"
#include "rng.h"
#include "aux_functions.h"

/*! \brief OptAlg class contains the functions that can be used to contruct an
optimization algorithm in the main function.
    The object needs the pointer to the problem object and the RNG object in order
to instantiate.
*/

class OptAlg {
    public:
        OptAlg() {};
        OptAlg(Problem *problem_ptr, Rng *gaussian_rng, int pop_size):
gaussian_rng(gaussian_rng) {
            prob = problem_ptr; //Store the pointer to the problem object.
            num = prob->num; //Store the number of variables.
            num_fit = prob->num_fit; //Store the number of fitness values.

            MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
            MPI_Comm_size(MPI_COMM_WORLD, &nb_proc);
            total_pop = pop_size; //Store the population size.
            }
        virtual ~OptAlg() {
            delete[] pop;
            };

        /*List of functions which are specific to the optimization functions*/
        virtual void put_to_best() {}; /*!< This function main purpose is to
prepare the population for optimization
                                        *after the population is initialized and
the first calculation of the fitness function.*/
        virtual void combination() {}; /*!< This function generates new candidates
that competes with the existing population.
                                        * This is one potential bottleneck in the
communication between processors.*/
        virtual void selection() {}; /*!< This function performs selects the
candidates for the next round of iteration.*/
        virtual void write_param(double *param_array) {}; /*!< This function let
the user specify new values to the search parameters.*/
```

```
        virtual void read_param(double *param_array) {}; /*!< This function reads
the search parameters that are currently in use.*/
        virtual void fit_to_global() {}; /*!< This function copies memories on
best array to global array.*/
        virtual void find_global() {}; /*!< This function finds global best
solution within a subset of the population and copy it to the global array.*/


        Problem* prob;

        //functions and variables that are not algorithm specific
        void Init_population(int psize); /*!< A function to initialize the
population using a uniform probability distribution.*/
        void Init_previous(double prev_dev, double new_dev, int psize, double
*prev_soln); /*!< A function to initialize the population using the solution from
N-1 variables.*/

        void Cont_fitness(int p); /*!< A function to compute the fitness value of
the solution in contender array.*/
        void Best_fitness(int p); /*!< A function to compute the fitness value of
the solution in best array.*/
        void update_popfit(); /*!< A function to compute the fitness value of the
solution in best array and update the mean fitness value.*/

        void set_success(int iter, bool goal); /*!<A function to set the condition
for which the optimization algorithm would accept the solution and stop.*/
        bool check_success(int t, double *current_fitarray, double
*memory_fitarray, int data_size, double t_goal, bool *mem_ptype, int *numvar, int
N_cut, double *memory_forT); /*!< A function for checking whether the current
solution/iteration satisfied the accept condition.*/

        //Selecting solution
        double Final_select(double *fit, double *solution, double *fitarray); /*!<
A function to find the candidate with the highest fitness value in the
population.*/
        double avg_Final_select(double* solution, int repeat, double *soln_fit,
double *fitarray); /*!< A function to compute the fitness function for 10 times
and find the candidate with the highest fitness value in the population*/

        void dev_gen(double *dev_array, double prev_dev, double new_dev, int
cut_off); /*!< A function for generating the array that indicates the width of the
Gaussian distribution for each of the variable.

* This array is used by Init_previous(). */
        int find_max(double *fit); /*!< A function to find the candidate that has
the highet fitness value
                                    * (for only one of the many fitness values
that a candidate posseses).*/

        bool success; /*!< Variable for indicating whether a solution is
accepted.*/
        bool policy_type; /*!< Variable for indicating whether the policy can be
used. */
```

```
    protected:
        int num; /*!< Number of variables*/
        int num_fit; /*!< Number of fitness values*/
        Rng *gaussian_rng; /*!< RNG object pointer*/
        int pop_size; /*!< The number of candidates on a processor.*/
        int T; /*!< The goal in number of iteration.*/
        int t; /*!<Time variables*/
        Candidate *pop; /*!< Pointer to population.*/
        bool goal; /*!< The variables indicating whether an accept-reject criteria
is used.*/

        int total_pop; /*!< The total number of candidates in the population.*/
        int my_rank; /*!< Variable for MPI*/
        int nb_proc;/*!< Variables for MPI*/

    };

/*! \brief DE class contains the functions that are specific to this particular
optimization algorithm.*/

class DE : public OptAlg
{
        public:
                DE() {};
                DE(Problem* problem_ptr, Rng* gaussian_rng, int pop_size) :
OptAlg(problem_ptr, gaussian_rng, pop_size), F(0.1), Cr(0.6) {};
                ~DE() {};

                void put_to_best();
                void combination();
                void selection();
                void fit_to_global();
                void find_global() {};
                void write_param(double* param_array);
                void read_param(double* param_array);

        private:
                double F; /*!< The search variable used in combining the base with
the differential part.*/
                double Cr; /*!< The crossover rate.*/

                void family_gen(int* fam, int p, int fam_size); /*!< This function
generates three random numbers indicating the candidates

* that are used by combination() to generate the next generation of candidate.*/

};

/*! \brief PSO class contains the functions that are specific to this particular
optimization algorithm.*/

class PSO : public OptAlg {
    public:
        PSO() {};
        PSO(Problem *problem_ptr, Rng *gaussian_rng, int pop_size):
OptAlg(problem_ptr, gaussian_rng, pop_size), w(0.8), phi1(0.6), phi2(1.0),
v_max(0.2) {};
        ~PSO() {};

        void put_to_best();
        void combination();
        void selection();
        void write_param(double *param_array);
        void read_param(double *param_array);
        void fit_to_global() {};
        void find_global();

    private:
        inline void find_index(int *prev, int *forw, int p); /*!< A function for
finding the index of neighbors of a candidate.*/
        inline int find_fitness(int prev, double prev_fit, int forw, double
forw_fit, int p, double fit); /*!< A functoin for getting the index of the
candidate in the neighborhood that has the highest fitness value.*/
        double w; /*!< The inertia variable to be multipled to the velocity.*/
        double phi1; /*!< The local search variable.*/
        double phi2; /*!< The global search variable.*/
        double v_max; /*!< The maximum speed for every variable.*/
    };

#endif // OPTALG_H
```

mpi_pso.cpp

```
#include "mpi_optalg.h"

void PSO::read_param(double* param_array) {
        param_array[0] = w;
        param_array[1] = phi1;
        param_array[2] = phi2;
        param_array[3] = v_max;
}

void PSO::write_param(double* param_array) {
        w = param_array[0];
        phi1 = param_array[1];
        phi2 = param_array[2];
        v_max = param_array[3];
}

void PSO::find_global() {
        /*! The global best position is find for a neighborhood of three. Because
of the circular topology,
        * the neighbors are on other processors and the fitness values from the
best array are sent back to the candidate.
        * The candidate then communicates with the processor containing the
highest fitness values of the three
        * and ask for the rest of the information.
        */
        MPI_Status status;
        int tag = 1;    // tag used for MPI comm
        // ptr. Best candidate among three.
        // prev, forw. The left and right candidate of the now candidate
        int ptr, prev, forw;
        double prev_fit, forw_fit, fit;
        double* fitarray = new double[this->prob->num_fit];
        double* array = new double[this->num];  // soln

        for (int p = 0; p < total_pop; ++p)
        {

                //find index of the candidate in the neighborhood
                find_index(&prev, &forw, p);

                //neighbor send fitness to the processor that contains candidate p
                // if I am prev, send fit to p
                if (my_rank == prev % nb_proc) {
                        prev_fit = this->pop[prev / nb_proc].read_bestfit(0);
                        MPI_Send(&prev_fit, 1, MPI_DOUBLE, p % nb_proc, tag,
MPI_COMM_WORLD);
                }
                // if I am p, send fit to prev
                else if (my_rank == p % nb_proc) {
                        MPI_Recv(&prev_fit, 1, MPI_DOUBLE, prev % nb_proc, tag,
MPI_COMM_WORLD, &status);
                }
                else {}

                // if I am forw, send fit to p
                if (my_rank == forw % nb_proc) {
                        forw_fit = this->pop[forw / nb_proc].read_bestfit(0);
                        MPI_Send(&forw_fit, 1, MPI_DOUBLE, p % nb_proc, tag,
MPI_COMM_WORLD);
                }
                // if I am p, send fit to forw
                else if (my_rank == p % nb_proc) {
                        MPI_Recv(&forw_fit, 1, MPI_DOUBLE, forw % nb_proc, tag,
MPI_COMM_WORLD, &status);
                }
                else {}

                MPI_Barrier(MPI_COMM_WORLD);

                //compare fitness at the processor and store the candidate with
best fitness in ptr
                // And send the best fitness to prev and forw
                if (my_rank == p % nb_proc) {
                        fit = this->pop[p / nb_proc].read_bestfit(0); //read
fitness of p
                        ptr = find_fitness(prev, prev_fit, forw, forw_fit, p,
fit);
                        //send ptr to prev and forw
                        MPI_Send(&ptr, 1, MPI_INT, prev % nb_proc, tag,
MPI_COMM_WORLD);
                }
                else if (my_rank == prev % nb_proc) {
                        MPI_Recv(&ptr, 1, MPI_INT, p % nb_proc, tag,
MPI_COMM_WORLD, &status);
                }
                else {}

                if (my_rank == p % nb_proc) {
                        MPI_Send(&ptr, 1, MPI_INT, forw % nb_proc, tag,
MPI_COMM_WORLD);
                }
                else if (my_rank == forw % nb_proc) {
                        MPI_Recv(&ptr, 1, MPI_INT, p % nb_proc, tag,
MPI_COMM_WORLD, &status);
                }
                else {}

                MPI_Barrier(MPI_COMM_WORLD);

                //updating global best
                if (ptr == p) { //global best already in processor's memory. No
need for MPI
```

```
                        if (my_rank == ptr % nb_proc) {
                                this->pop[p / nb_proc].put_to_global();
                        }
                        else {}
                }
                else if (ptr == prev || ptr == forw) {
                        if (my_rank == ptr % nb_proc) {
                                this->pop[ptr / nb_proc].read_best(array);
                                MPI_Send(&array[0], this->num, MPI_DOUBLE, p %
nb_proc, tag, MPI_COMM_WORLD);
                        }
                        else if (my_rank == p % nb_proc) {
                                MPI_Recv(&array[0], this->num, MPI_DOUBLE, ptr %
nb_proc, tag, MPI_COMM_WORLD, &status);
                                this->pop[p / nb_proc].update_global(array);
                        }
                        //sending the fitarray
                        if (my_rank == ptr % nb_proc) {
                                for (int i = 0; i < this->num_fit; ++i) {
                                        fitarray[i] = this->pop[ptr /
nb_proc].read_globalfit(i);
                                }
                                MPI_Send(&fitarray[0], this->num_fit, MPI_DOUBLE,
p % nb_proc, tag, MPI_COMM_WORLD);
                        }
                        else if (my_rank == p % nb_proc) {
                                MPI_Recv(&fitarray[0], this->num_fit, MPI_DOUBLE,
ptr % nb_proc, tag, MPI_COMM_WORLD, &status);
                                this->pop[p / nb_proc].write_globalfit(fitarray);
                        }
                        else {}
                }
                else {}
                //end updating global best

        }//p loop

        delete[] fitarray;
        delete[] array;

}
inline void PSO::find_index(int* prev, int* forw, int p) {
        /*! Finding the indexes of candidate in the neighborhood. We use the
circular topology in order to determine the neighborhood.
        */
        if (p == 0) {
                *prev = total_pop - 1;
        }
        else {
                *prev = p - 1;
        }
        *forw = (p + 1) % total_pop;
}
```

```
inline int PSO::find_fitness(int prev, double prev_fit, int forw, double forw_fit,
int p, double fit) {
        /*! Find the candidate with the best fitness value from a group of three.
        */
        int ptr = prev;
        if (prev_fit <= fit) {
                ptr = p;
                if (fit < forw_fit) {
                        ptr = forw;
                }
                else {}   //ptr=p
        }
        else if (prev_fit > fit) {
                if (prev_fit < forw_fit) {
                        ptr = forw;
                }
                else {}   //ptr still points to prev
        }
        else {}
        return ptr;
}

void PSO::put_to_best() {
        /*! This function in PSO record the first position as personal best,
initialize the velocity and find global best position
        * in preparation for the iterative steps.
        */
        double* array = new double[this->num];

        for (int p = 0; p < this->pop_size; ++p) {
                this->pop[p].update_best();
                this->pop[p].init_velocity();
                //generating velocity
                for (int i = 0; i < this->num; ++i) {
                        array[i] = double(rand()) / RAND_MAX * v_max;
                }
                this->pop[p].update_vel(array);
        }

        find_global();

        delete[] array;

}

void PSO::selection() {
        /*!In PSO, selection chooses whether the position stored in contender is a
new personal best,
        * and, if so, store it in best array. The global best position is then
searched and updated.
        */
        for (int p = 0; p < this->pop_size; ++p) {
                if (this->pop[p].read_bestfit(0) < this->pop[p].read_contfit(0)) {
                        this->pop[p].update_best();
                }
                else {}
        }
        find_global();
}

void PSO::combination() {
        /*! Combination() generates the position and velocity for the next time
step using the rule with inertia term.
        * The new position is stored in the contender and computed for mean
fitness value.
        */
        double* global_pos = new double[this->num];
        double* personal_pos = new double[this->num];
        double* pos = new double[this->num];
        double* vel = new double[this->num];
        double* new_pos = new double[this->num];

        for (int p = 0; p < this->pop_size; ++p) {
                this->pop[p].read_global(global_pos);
                this->pop[p].read_best(personal_pos);
                this->pop[p].read_cont(pos);
                this->pop[p].read_vel(vel);
                for (int i = 0; i < this->num; ++i) {
                        new_pos[i] = pos[i] + vel[i];
                        vel[i] = w * vel[i] + phi1 * double(rand()) / RAND_MAX *
(personal_pos[i] - pos[i]) + phi2 * double(rand()) / RAND_MAX * (global_pos[i] -
pos[i]);
                        if (vel[i] > v_max) {
                                vel[i] = v_max;
                        }
                        else if (vel[i] < -v_max) {
                                vel[i] = -v_max;
                        }
                        else {}
                }
                this->prob->boundary(new_pos);  //keep the solution within the
boundary of the search space
                this->pop[p].update_cont(new_pos);      // update the new position
                this->pop[p].update_vel(vel);   //update the new velocity
                this->Cont_fitness(p);  // update the new fitness value
        }

        delete[] global_pos;
        delete[] personal_pos;
        delete[] pos;
        delete[] vel;
        delete[] new_pos;

}
```

mpi_de.cpp

```
#include "mpi_optalg.h"

void DE::put_to_best() {
    for(int p = 0; p < this->pop_size; ++p) { //pop_size here is the number of
candidates assigned for a processor from the initialization.
        this->pop[p].update_best();
        }
    }

void DE::write_param(double *param_array) {
    F = param_array[0];
    Cr = param_array[1];
    }

void DE::read_param(double *param_array) {
    param_array[0] = F;
    param_array[1] = Cr;
    }

void DE::fit_to_global() {
    for(int p = 0; p < this->pop_size; ++p) {
        this->pop[p].put_to_global();
        }
    }

void DE::selection() {
    /*! Selection() reads the mean fitness value of best array and contender and
decides which one is to be the member of the population.
    In this version, we simple select the one with the higher fitness value.
    */
    for(int p = 0; p < this->pop_size; ++p) {
        if(this->pop[p].read_bestfit(0) < this->pop[p].read_contfit(0)) {
            this->pop[p].update_best();
            }
        }
    }

void DE::combination()
{
        /*! Combination() generates the next generation of candidates. First, all
the candidate give their best array to the zeroth processor
        * who then generates the new candidate and send it back to the parent
candidate. The parent store the new array into the contender
        * and compute the mean fitness value.
        */
        MPI_Status status;
        int tag = 1;
        double coin;
        int fam_size = 3;

        double** all_soln = new double* [total_pop]; // [this->num] ;
        for (int i = 0; i < total_pop; i++)
                all_soln[i] = new double[this->num];
        int* fam = new int [fam_size];
        double* input = new double [this->num];

        //get the solution from all processor to root ***POTENTIAL BOTTLENECK***
        for (int p = 0; p < total_pop; ++p) {
                if (p % nb_proc != 0) { //if candidate is not in root, send the
solution to root.
                        if (my_rank == p % nb_proc) {
                                this->pop[int(p / nb_proc)].read_best(input);
                                MPI_Send(&input, this->num, MPI_DOUBLE, 0, tag,
MPI_COMM_WORLD);
                        }
                        else if (my_rank == 0) {
                                MPI_Recv(&input, this->num, MPI_DOUBLE, p %
nb_proc, tag, MPI_COMM_WORLD, &status);
                                for (int i = 0; i < this->num; ++i) {
                                        all_soln[p][i] = input[i];
                                }
                        }
                }
                else if (p % nb_proc == 0) { //if candidate is in root, read the
solution into the memory
                        if (my_rank == 0) {
                                this->pop[int(p / nb_proc)].read_best(input);
                                for (int i = 0; i < this->num; ++i) {
                                        all_soln[p][i] = input[i];
                                }
                        }
                }
        }// p loop

        MPI_Barrier(MPI_COMM_WORLD);

        //generate the new candidate
        for (int p = 0; p < total_pop; ++p) {
                if (my_rank == 0) {
                        family_gen(fam, p, fam_size);
                        //create donor
                        for (int i = 0; i < this->num; ++i) {
                                //create donor
                                coin = double(rand()) / RAND_MAX;
                                if (coin <= Cr || i == rand() % this->num) {
                                        input[i] = all_soln[fam[0]][i] + F *
(all_soln[fam[1]][i] - all_soln[fam[2]][i]);
                                }
                                else {
                                        input[i] = all_soln[p][i];
                                }
                        }
                        this->prob->boundary(input);

                }//my_rank==0

        //update of candidate
                if (p % nb_proc == 0) { // if p candidate is in root, update the
candidate
                        if (my_rank == 0) {
                                this->pop[int(p /
nb_proc)].update_cont(&input[0]);
                        }
                }
                else { // if p candidate is not on root, send the new candidate
from root to processor
                        if (my_rank == 0) {
                                MPI_Send(&input[0], this->num, MPI_DOUBLE, p %
nb_proc, tag, MPI_COMM_WORLD);
                        }
                        else if (my_rank == p % nb_proc) { //processor that
contains p candidate updates the candidate
                                MPI_Recv(&input[0], this->num, MPI_DOUBLE, 0, tag,
MPI_COMM_WORLD, &status);
                                this->pop[int(p /
nb_proc)].update_cont(&input[0]);
                        }
                        else {}
                }

        }//p loop

        MPI_Barrier(MPI_COMM_WORLD);

        //all candidate calculate fitness of contender
        for (int p = 0; p < this->pop_size; ++p) {
                this->Cont_fitness(p);   //compute the fitness
        }

        MPI_Barrier(MPI_COMM_WORLD);

        for (int i = 0; i < total_pop; i++)
                delete[] all_soln[i];
        delete[] all_soln;

        delete[] fam;
        delete[] input;
}

void DE::family_gen(int* fam, int p, int fam_size) {
    /*! Three candidates are chosen at random from the population are selected to
generate the next generation.
    * The candidates selected for the family are not allow to be the parent or the
same individual more than once.
    */
    if(total_pop <= fam_size + 1) {
        for(int f = 0; f < fam_size; ++f) {
            fam[f] = rand() % total_pop;
            }
        }
    else {
        do fam[0] = rand() % total_pop;
        while(fam[0] == p);
        do fam[1] = rand() % total_pop;
        while(fam[1] == p || fam[1] == fam[0]);
        do fam[2] = rand() % total_pop;
        while(fam[2] == p || fam[2] == fam[0] || fam[2] == fam[1]);
        }
    }
```

phase_loss_opt.cpp

```
//#include <iostream>
//#include <cstdlib>
//#include <cstring>
//#include <stdexcept>
//#include <algorithm>
//#include "corecrt_math_defines.h"
//
//
//#include "phase_loss_opt.h"

//using namespace std;


//Phase::Phase(const int numvar_f, Rng* gaussian_rng_f, Rng* uniform_rng_f) //:
numvar(numvar), gaussian_rng(gaussian_rng), uniform_rng(uniform_rng)
//{
//      /*! The instantiation function of the class initialize necessary variables
that the optimization algorithm uses.
//      * Memories used by the problem is also allocated in this stage.
//      */
//      numvar = numvar_f;
//      gaussian_rng = gaussian_rng_f;
//      uniform_rng = uniform_rng_f;
//
//      if (numvar <= 0) {
//              throw invalid_argument("numvar<=0. Instantiating Phase fails.");
//      }
//
//      //Initializing the conditions the simulation uses.
//      lower = 0;
//      upper = 2 * M_PI;
//      loss = 0.0;
//
//      //Initializing the numbers used by the optimization algorithm.
//      num = numvar;
//      num_fit = 2;
//      num_repeat = 10 * num * num;
//
//      //Initializing the lower bound and upper bound for the optimized
variables.
//      lower_bound = new double[num];
//      upper_bound = new double[num];
//      for (int i = 0; i < num; ++i) {
//              lower_bound[i] = lower;
//              upper_bound[i] = upper;
//      }
//      //Initializing memories used by the simulation.
//      sqrt_cache = new double[num + 1];
//      for (int i = 0; i < num + 1; ++i) {
//              sqrt_cache[i] = sqrt(i);
//      }
//      input_state = new dcmplx[num + 1];
//      sqrtfac_mat = new double[num + 1];
//      overfac_mat = new double[num + 1];
//      state = new dcmplx[num + 1];
//      update0 = new dcmplx[num + 1];
//      update1 = new dcmplx[num + 1];
//}

//Phase::~Phase() {
//      /*freeing the memory.
//      * This is a crucial step if the main function runs a loop with multiple
instantiation of the phase problem
//      as the computer might run out of memory.
//      */
//      delete[] state;
//      delete[] update0;
//      delete[] update1;
//      delete[] input_state;
//      delete[] sqrtfac_mat;
//      delete[] overfac_mat;
//}

//void Phase::fitness(double* soln, double* fitarray) {
//      /*In this particular problem, this function serves as a wrapper to change
the loss rate,
//      * so that we can test the policy we found in lossless interferometry with
a chosen level of loss.
//      */
//      loss = 0.2; //This loss rate can be changed by user.
//      avg_fitness(soln, num_repeat, fitarray);
//      loss = 0.0; //Change back in case the optimization process has to be
redone.
//}
//
//void Phase::avg_fitness(double* soln, const int K, double* fitarray) {
//      /* A function calculates the fitness values (reported in fitarray) of a
solution (soln) over a sample size of K.
//      * This function simulates the adaptive phase interferometry and so is the
most computationally expensive part of the program.
//      */
//      dcmplx sharp(0.0, 0.0); //variable to store the sharpness function, which
is the first fitness value in the array.
//      double error = 0.0; //variable to store the bias of the estimate, which is
the second fitness value in the array.
//      bool dect_result; //variable to store which path a photon comes out at any
step.
//      double PHI, phi, coin, PHI_in;
//      int m, k, d;
//
//      WK_state(); //Generate the WK state.
//
```

```
//      for (k = 0; k < K; ++k) {
//              phi = uniform_rng->next_rand(0.0, 1.0) * (upper - lower) + lower;
//              PHI = 0;
//              //copy input state: the optimal solution across all compilers is
memcpy:
//
//nadeausoftware.com/articles/2012/05/c_c_tip_how_copy_memory_quickly
//              memcpy(state, input_state, (num + 1) * sizeof(dcmplx));
//              //Begining the measurement of one sample
//              d = 0;
//              for (m = 0; m < num; ++m) {
//                      // This loop is the most critical part of the entire
program. It
//                      // executes K*num=10*num^3 times on each call of
avg_fitness. All
//                      // optimization should focus on this loop.
//
//                      //randomly decide whether loss occurs
//                      coin = uniform_rng->next_rand(0.0, 1.0);
//
//                      if (coin <= loss) {
//                              state_loss(num - m); //update only the state using
loss function
//                      }
//                      else {
//                              //PHI_in = rand_Gaussian(PHI, THETA_DEV); //select
if the noise is normally distributed.
//              //PHI_in = rand_Hat(PHI, THETA_DEV); //select if the noise is
distributed as a Hat function.
//              //PHI_in = Lognormal(MU,THETA,PHI); //select if the noise is
distributed as a lognormal function.
//              //PHI_in = rand_RTN(PHI,Ps,THETA_DEV);//select if the noise is
random telegraph.
//                              PHI_in = rand_skewed(PHI, THETA_DEV, RATIO);
//select if the noise is skewed normal.
//                              PHI_in = mod_2PI(PHI_in);//noisy PHI
//                              dect_result = noise_outcome(phi, PHI_in, num - m);
//                              if (dect_result == 0) {
//                                      PHI = PHI - soln[d++];
//                              }
//                              else {
//                                      PHI = PHI + soln[d++];
//                              }
//                              PHI = mod_2PI(PHI);
//                      }
//              }
//              //store fitness values
//              sharp.real(sharp.real() + cos(phi - PHI));
//              sharp.imag(sharp.imag() + sin(phi - PHI));
//      }
//      //find the averages and return
//      fitarray[0] = abs(sharp) / double(K); //Calculate the dispersion
//      fitarray[1] = atan2(sharp.imag(), sharp.real()); //Calculate the mean
direction
```

```
//}
//
//bool Phase::T_condition(double* fitarray, int* numvar, int N_cut, bool*
mem_ptype, double* memory_forT) {
//      /*This function contains the conditions that has to be checked after a
step T elapses
//      * before the algorithm decides to accept or reject the solution.
//      * In particular this function is called when time step is used as the main
condition to end the optimization.
//      * For this particular problem, it resets the number of variables so the
optimization starts over.
//      */
//      //Let's start this condition from scratch.
//      bool type;
//
//      //The conditions are checked only if the algorithm is going to change the
way it initializes the population.
//      //Policy type 1 is the one with pi bias and is susceptible to loss. CThe
algorithm will run until type 0 is found.
//      if (*numvar == N_cut - 1) {
//              try {
//                      type = check_policy(fitarray[1], fitarray[0]);
//              }
//              catch (invalid_argument) {
//                      fitarray[0] = 0.999999;
//                      type = check_policy(fitarray[1], fitarray[0]);
//              }
//              mem_ptype[0] = type;
//              if (type == 1) {
//                      *numvar = *numvar - 1;
//              }
//              else if (type == 0) {
//                      memory_forT[0] = fitarray[0];
//                      memory_forT[1] = fitarray[1];
//              }
//              else {}
//      }
//      else if (*numvar == N_cut) {
//              try {
//                      type = check_policy(fitarray[1], fitarray[0]);
//              }
//              catch (invalid_argument) {
//                      fitarray[0] = 0.999999;
//                      type = check_policy(fitarray[1], fitarray[0]);
//              }
//              mem_ptype[1] = type;
//              //        if(mem_ptype[0] | type) {
//              if (check_policy(memory_forT[1], memory_forT[0]) | type) {
//                      //the policy is bad
//                      //reset the policy found in numvar=N_cut-1
//                      *numvar = N_cut - 2;
//              }
//      }
//      return 1;
```

```
//}
//
//bool Phase::error_condition(double* current_fitarray, double* memory_fitarray,
int data_size, double goal) {
//      /*This function contains the function to compute error for when the
algorithm is set to use error as the accept-reject condition.
//      *It allows for the information of previous optimization to be used to
compute the condition as well as using the latest result.
//      *In this particular problem, we use the error that corresponds to the
confidence interval of 0.98
//      * from the linear relation between logN and logV_H.
//      */
//
//      double slope, intercept;
//      double mean_x;
//      double tn2;
//      double error, error_goal;
//
//      int xy_data_size = data_size + 1;
//
//      double* x = new double[xy_data_size];
//      double* y = new double[xy_data_size];
//
//      bool out;
//
//      memory_fitarray[2 * (data_size)] = log10(num);
//      memory_fitarray[2 * (data_size)+1] = log10(pow(current_fitarray[0], -2) -
1);
//
//      //split into x-y arrays
//
//      for (int i = 0; i <= data_size; ++i) {
//              x[i] = memory_fitarray[2 * i];
//              y[i] = memory_fitarray[2 * i + 1];
//      }
//
//      //Computing the linear equation using previous data
//      linear_fit(data_size, x, y, &slope, &intercept, &mean_x);
//      //Compute the goal for error from confidence interval of 0.98
//      error_goal = error_interval(x, y, mean_x, data_size + 1, slope,
intercept);
//
//      goal = (goal + 1) / 2;
//      tn2 = quantile(goal);
//      error_goal = error_goal * tn2;
//
//      //Compute the distance between the data and the prediction from linear
equation
//      error = y[data_size] - x[data_size] * slope - intercept;
//
//      //Check if error is smaller than the goal
//      if (error <= error_goal) {
//              out = 1;
//      }
//      else {
//              out = 0;
//      }
//      return out;
//}
//
//
//
//void Phase::boundary(double* can1) {
//      /*This function cuts the value of the variables so they are within the
boundary of the problem.
//      *This can be replaced by one or more method, such as periodic boundary or
normalization.
//      */
//      for (int i = 0; i < num; ++i) {
//              if (can1[i] < lower_bound[i]) {
//                      can1[i] = lower_bound[i];
//              }
//              else if (can1[i] > upper_bound[i]) {
//                      can1[i] = upper_bound[i];
//              }
//      }
//}
//
///*private functions: specific to the simulation and not used by the optimization
algorithm*/
//
///*########### Generating Input State ###########*/
///*Generation function*/
//void Phase::WK_state() {
//      /* This is the main function for generating the input state.
//      * We call it here the Wiseman-Killip state, but the more general term
would be the sine state.
//      * This algorithm for generating WK state is the most precise one to date
(measured by how much the normalization differs from zero),
//      * but the rounding error still appears as early as N=80 and is detrimental
to the shape of the state for N>100.
//      * This is a limitation caused by the use of double precision float.
//      * For N>100, either switch to quadruple precision, which will cause a
large overhead
//      * (about 15 for the simulation, but even more for this function), or
another method has to be used to approximate the state.
//      */
//      const double beta = M_PI / 2;
//      const double cosN = pow(cos(beta / 2), num);
//      tan_beta = tan(beta / 2);
//
//      int n, k;
//      double* n_part = new double[num + 1];
//      double* k_part = new double[num + 1];
//      double s_part;
//      double temp;
//
//      //Preparing constant arrays to save time.
```

```
//
//      //initializing array of factorial numbers
//      sqrtfac(sqrtfac_mat);
//      one_over_fac(overfac_mat);
//
//      /*factors in d_matrix calculated*/
//      //calculating n_parts and k_parts in
//      for (n = 0; n <= num; ++n) {
//              n_part[n] = pow(-1.0, n) * sqrtfac_mat[n] * sqrtfac_mat[num - n] *
cosN * pow(tan_beta, n);
//      }
//      for (k = 0; k <= num; ++k) {
//              k_part[k] = 1 / pow(-1.0, k) * sqrtfac_mat[k] * sqrtfac_mat[num -
k] * pow(1 / tan_beta, k);
//      }
//
//      //Compute the state
//      for (n = 0; n <= num; ++n) { //we have N+1 state b/c we include n=0 and
n=N.
//              temp = 0;
//              input_state[n].real(0);
//              input_state[n].imag(0);
//              for (k = 0; k <= num; ++k) {
//                      s_part = cal_spart(n, k, num);
//                      temp = s_part * k_part[k] * sin((k + 1) * M_PI / (num +
2));
//                      input_state[n].real(input_state[n].real() + temp *
cos(M_PI / 2.0 * (k - n)));
//                      input_state[n].imag(input_state[n].imag() + temp *
sin(M_PI / 2.0 * (k - n)));
//              }//end k
//              input_state[n].real(input_state[n].real() * n_part[n] / sqrt(num /
2.0 + 1));
//              input_state[n].imag(input_state[n].imag() * n_part[n] / sqrt(num /
2.0 + 1));
//      }//end n
//
//      delete[] n_part;
//      delete[] k_part;
//
//}
//
//inline double Phase::cal_spart(const int n, const int k, const int N) {
//      /*This function calculates the sum in an element of a Wigner d-matrix
//      */
//
//      int s;
//      int s_min;
//      int s_max;
//      double s_part = 0;
//
//      //find lower limit
//      if (n - k >= 0) {
//              s_min = 0;
//      }
//      else if (n - k < 0) {
//              s_min = k - n;
//      }
//      else {}
//      //find upper limit
//      if (k <= N - n) {
//              s_max = k;
//      }
//      else if (k > N - n) {
//              s_max = N - n;
//      }
//      else {}
//
//      //calculating the s_part
//      for (s = s_min; s <= s_max; ++s) {
//              s_part = s_part + pow(-1.0, s) * overfac_mat[k - s] *
overfac_mat[s] * overfac_mat[n - k + s] * overfac_mat[N - n - s] * pow(tan_beta, 2
* s);
//      }
//      return s_part;
//}
//
//inline void Phase::one_over_fac(double* over_mat) {
//      /*This function calculates one over factorial matrix.*/
//      over_mat[0] = 1;
//      for (int i = 1; i <= num; ++i) {
//              over_mat[i] = over_mat[i - 1] / i;
//      }//end i
//}
//
//inline void Phase::sqrtfac(double* fac_mat) {
//      /*This function calculates sqrare root of factorial matrix.*/
//      //check array size
//      fac_mat[0] = 1;
//      for (int i = 1; i <= num; ++i) {
//              fac_mat[i] = fac_mat[i - 1] * sqrt_cache[i];
//      }//end i
//}
//
///*########### Measurement Functions ###########*/
///*The following functions are involved in the simulation of the noisy
interferometer.*/
//
//inline bool Phase::noise_outcome(const double phi, const double PHI, const int
N) {
//      /*This function computes the output path of a photon from a noisy
interferometer
//      by computing the probablity of photon coming out of either path.
//      This simulation allows for noise in the unitary operation, but we only
consider the noise in the phase shift.
//      */
//      //N is the number of photons currently available, not equal to 'num'
//      const double theta = (phi - PHI) / 2.0;
```

```
//	const double cos_theta = cos(theta) / sqrt_cache[N];
//	const double sin_theta = sin(theta) / sqrt_cache[N];
//	//noise in operator: currently not in use
//	const double oper_n0 = gaussian_rng->next_rand(0.0, DEV_N);//n_x
//	const double oper_n2 = gaussian_rng->next_rand(0.0, DEV_N);//n_z
//	const double oper_n1 = sqrt(1.0 - (oper_n0 * oper_n0 + oper_n2 *
oper_n2));
//	const dcmplx U00(sin_theta * oper_n1, -oper_n0 * sin_theta);
//	const dcmplx U01(cos_theta, oper_n2 * sin_theta);
//	const dcmplx U10(cos_theta, -oper_n2 * sin_theta);
//	const dcmplx U11(sin_theta * oper_n1, sin_theta * oper_n0);
//	int n;
//	double prob = 0.0;
//	for (n = 0; n < N; ++n) {
//		//if C_0 is measured
//		update0[n] = state[n + 1] * U00 * sqrt_cache[n + 1] + state[n] *
U01 * sqrt_cache[N - n];
//		prob += abs(update0[n] * conj(update0[n]));
//	}
//
//	if (uniform_rng->next_rand(0.0, 1.0) <= prob) {
//		//measurement outcome is 0
//		state[N] = 0;
//		prob = 1.0 / sqrt(prob);
//		for (n = 0; n < N; ++n) {
//			state[n] = update0[n] * prob;
//		}
//		return 0;
//	}
//	else {
//		//measurement outcome is 1
//		prob = 0;
//		for (n = 0; n < N; ++n) {
//			state[n] = state[n + 1] * U10 * sqrt_cache[n + 1] -
state[n] * U11 * sqrt_cache[N - n];
//			prob += abs(state[n] * conj(state[n]));
//		}
//		state[N] = 0;
//		prob = 1.0 / sqrt(prob);
//		for (n = 0; n < N; ++n) {
//			state[n] *= prob;
//		}
//		return 1;
//	}
//}
//
//inline void Phase::state_loss(const int N) {
//	/*This function updates the state when one of the photon is loss.*/
//	double total = 0;
//	double factor = 1 / sqrt(2 * N);
//	for (int n = 0; n < N; ++n) {
//		state[n] = (state[n] * sqrt_cache[N - n] + state[n + 1] *
sqrt_cache[n + 1]) * factor;
//		total += state[n].real() * state[n].real() + state[n].imag() *
```

```
state[n].imag();
//      }
//      state[N] = 0;
//      //Necessary state renormalization
//      for (int n = 0; n < N; ++n) {
//              state[n] = state[n] / sqrt(total);
//      }
//}
//
//inline double Phase::mod_2PI(double PHI) {
//      /*This function compute the modulo of the phase.
//      */
//      while (PHI >= 2 * M_PI) {
//              PHI = PHI - 2 * M_PI;
//      }
//      while (PHI < 0) {
//              PHI = PHI + 2 * M_PI;
//      }
//      return PHI;
//}
//
//inline bool Phase::check_policy(double error, double sharp) {
//      /*This function takes the bias and output the policy type.
//      *A policy is considered to be type zero if its error falls within the
uncertainty of the scheme.
//      *This is the desirable type as its estimate no bias and the policy can be
used when loss is presence.
//      *In the error falls outside the uncertainty,
//      *it is very likely that the estimate has a pi bias and is the type of
policy that fails when there is loss.
//      */
//      if (sharp == 1.0) {
//              throw invalid_argument("sharpness cannot be one.");
//      }
//      double sd = sqrt(1 / (sharp * sharp) - 1);
//      if (fabs(error) >= fabs(M_PI - sd)) {
//              return 1;
//      }
//      else {
//              return 0;
//      }
//}
//
//double Phase::rand_Gaussian(double mean, /*the average theta*/
//      double dev /*deviation for distribution*/
//) {
//      /*creating random number using Box-Muller Method/Transformation*/
//      double Z0;//,Z1;
//      double U1, U2; /*uniformly distributed random number input*/
//      double r;
//
//      /*create input between [-1,1]*/
//      do {
//              U1 = 2.0 * double(rand()) / RAND_MAX - 1.0;
```

```
//		U2 = 2.0 * double(rand()) / RAND_MAX - 1.0;
//		r = U1 * U1 + U2 * U2;
//	} while (r == 0.0 || r >= 1.0);
//	/*using Box-Muller Transformation*/
//	Z0 = U1 * sqrt(-2 * log(r) / r);
//
//	return Z0 * dev + mean;
//}/*end of rand_Gaussian*/
//
//double Phase::rand_Hat(double PHI, double dev) {
//	return gaussian_rng->next_rand(PHI, dev);
//}
//
//inline double Phase::inv_erf(double x) {
//	if (x == 1) {
//		throw invalid_argument("Input leads to error.");
//	}
//	double a = 0.140012;
//	double lnx = log(1 - x * x);
//	double temp = sqrt(pow(2.0 / (M_PI * a) + lnx / 2.0, 2) - lnx / a);
//
//	return sgn(x) * sqrt(temp - 2.0 / (M_PI * a) - lnx / 2.0);
//}
//
//inline int Phase::sgn(double x) {
//	/** Sign function of x: https://en.wikipedia.org/wiki/Sign_function
//	*/
//	if (x < 0) {
//		return -1;
//	}
//	else if (x == 0) {
//		return 0;
//	}
//	else {
//		return 1;
//	}
//}
//
//double Phase::Lognormal(double mu, double sigma, double peak) {
//	double mode = exp(mu - sigma * sigma); //where the peak is, so we know how
much to move the distribution later.
//	double diff = mode - peak;
//	double ans;
//
//	double u = (double)(rand()) / ((double)(RAND_MAX));
//	double p = 2 * u - 1;
//
//	try {
//		ans = exp(sqrt(2) * sigma * inv_erf(p) + mu) - diff;
//	}
//	catch (invalid_argument) {
//		u = (double)(rand()) / ((double)(RAND_MAX));
//		p = 2 * u - 1;
//		ans = exp(sqrt(2) * sigma * inv_erf(p) + mu) - diff;
```

```
//      }
//
//      return ans;
//}
//
//double Phase::rand_RTN(double PHI, double ps, double dev) {
//      double coin = double(rand()) / double(RAND_MAX);
//      double ans;
//      if (coin <= ps) {
//              coin = double(rand()) / double(RAND_MAX);
//              if (coin < 0.5) {
//                      ans = PHI + dev;
//              }
//              else {
//                      ans = PHI - dev;
//              }
//      }
//      else {
//              ans = PHI;
//      }
//      return ans;
//}
//
//double Phase::rand_skewed(double mean, double dev, double ratio) {
//      double u1, u2;
//      double k1, k2;
//
//      double alpha = ratio * dev;
//
//      k1 = (1 + alpha) / sqrt(2 * (1 + alpha * alpha));
//      k2 = (1 - alpha) / sqrt(2 * (1 + alpha * alpha));
//
//      u1 = rand_Gaussian(0, 1);
//      u2 = rand_Gaussian(0, 1);
//
//      return (k1 * max(u1, u2) + k2 * min(u1, u2)) * dev + mean;
//}
```

phase_loss_opt.h

```
#ifndef PHASE_LOSS_H
#define PHASE_LOSS_H

#include <complex>
#if HAVE_CONFIG_H
#include <config.h>
#endif

#define DEV_N 0.0 //level of noise in operator
//Normal and Hat noise parameters
#define THETA_DEV 1.41421356237//M_PI;//phase noise level
//RTN parameter
#define Ps 0.5 //probability of telegraph noise
//SND parameter
#define RATIO 1.176959769 //ratio between alpha and sigma
//Lognormal parameters
#define MU 2.2214//variance
#define THETA 0.2715 //skewness, variance

#include <iostream>
#include <cstdlib>
#include <cstring>
#include <stdexcept>
#include <algorithm>
#include "corecrt_math_defines.h"

#include "problem.h"
#include "rng.h"
#include "aux_functions.h"

typedef complex<double> dcmplx;

/*! \brief Phase class for the problem of adaptive interferometric phase
estimation including noise and loss.
*
* This class can be replaced by any optimization problem of interest written by
the user.
* The problem is initialized in the main function through the Problem class
pointer.
*/

class Phase : public Problem
{
        public:
                Phase(const int numvar_f, Rng* gaussian_rng_f, Rng*
uniform_rng_f);
                ~Phase();
                void fitness(double* soln, double* fitarray);
                void avg_fitness(double* soln, const int K, double* fitarray);
                bool T_condition(double* fitarray, int* numvar, int N_cut, bool*
mem_ptype, double* memory_forT);
                bool error_condition(double* current_fitarray, double*
memory_fitarray, int data_size, double goal);
                void boundary(double* can1);

        private:
                double lower; //The lower bound of the variables
                double upper; //The upper bound of the variables
                double loss; //The photon loss rate

                int numvar;
                Rng* gaussian_rng, * uniform_rng;

                //variables for WK state generation
                dcmplx* input_state; //The array containing the WK state
                double* sqrtfac_mat; //matrix to keep values of square roots of
factorials
                double* overfac_mat; //matrix to keep values of one over
factorials
                double* sqrt_cache; //array to avoid calculation of expensive sqrt
calls for integers
                double tan_beta;

                dcmplx* state; //state for use with measurement
                dcmplx* update0; //variables for noise_output function
                dcmplx* update1;

                //functions to generate WK_state
                inline void sqrtfac(double* fac_mat); //A function for calculating
square root of factorials for state generation.
                inline void one_over_fac(double* over_mat); //A function for
calculating one over square root of factorials for state generation.
                inline double cal_spart(const int n, const int k, const int N);//N
is the same as total number of photon 'num'.
                void WK_state(); //A funtion generating the WK state
                //Measurement functions
                inline bool noise_outcome(const double phi, const double PHI,
const int N); //A function for simulating a photon going through a noisy Mach-
Zehnder interferometer.
                inline void state_loss(const int N); //A function for simulating
state change under loss of a photon.
                inline double mod_2PI(double PHI); //A function to perform modulo
2PI on phase.

                bool check_policy(double error, double sharp); //A function for
checking whether the policy is resilient to loss. This is called by the
T_condition().

                double rand_Gaussian(double mean, double dev);
                double rand_Hat(double PHI, double dev);
                //lognormal-distribution noise
                inline double inv_erf(double x); /*!< Function for calculating the
inverse of an error function.*/
                inline int sgn(double x); /*!< Sign function.*/
                double Lognormal(double mu, double sigma, double peak);
                //Random telegraph noise
                double rand_RTN(double PHI, double ps, double dev);
                //Skewed normal noise
                double rand_skewed(double mean, double dev, double ratio);
};

Phase::Phase(const int numvar_f, Rng* gaussian_rng_f, Rng* uniform_rng_f) //:
numvar(numvar), gaussian_rng(gaussian_rng), uniform_rng(uniform_rng)
{
        /*! The instantiation function of the class initialize necessary variables
that the optimization algorithm uses.
        * Memories used by the problem is also allocated in this stage.
        */
        numvar = numvar_f;
        gaussian_rng = gaussian_rng_f;
        uniform_rng = uniform_rng_f;

        if (numvar <= 0) {
                throw invalid_argument("numvar<=0. Instantiating Phase fails.");
        }

        //Initializing the conditions the simulation uses.
        lower = 0;
        upper = 2 * M_PI;
        loss = 0.0;

        //Initializing the numbers used by the optimization algorithm.
        num = numvar;
        num_fit = 2;
        num_repeat = 10 * num * num;

        //Initializing the lower bound and upper bound for the optimized
variables.
        lower_bound = new double[num];
        upper_bound = new double[num];
        for (int i = 0; i < num; ++i) {
                lower_bound[i] = lower;
                upper_bound[i] = upper;
        }
        //Initializing memories used by the simulation.
        sqrt_cache = new double[num + 1];
        for (int i = 0; i < num + 1; ++i) {
                sqrt_cache[i] = sqrt(i);
        }
        input_state = new dcmplx[num + 1];
        sqrtfac_mat = new double[num + 1];
        overfac_mat = new double[num + 1];
        state = new dcmplx[num + 1];
        update0 = new dcmplx[num + 1];
        update1 = new dcmplx[num + 1];
```

```
}

Phase::~Phase() {
        /*freeing the memory.
        * This is a crucial step if the main function runs a loop with multiple
instantiation of the phase problem
        as the computer might run out of memory.
        */
        delete[] state;
        delete[] update0;
        delete[] update1;
        delete[] input_state;
        delete[] sqrtfac_mat;
        delete[] overfac_mat;
}

void Phase::fitness(double* soln, double* fitarray) {
        /*In this particular problem, this function serves as a wrapper to change
the loss rate,
        * so that we can test the policy we found in lossless interferometry with
a chosen level of loss.
        */
        loss = 0.2; //This loss rate can be changed by user.
        avg_fitness(soln, num_repeat, fitarray);
        loss = 0.0; //Change back in case the optimization process has to be
redone.
}

void Phase::avg_fitness(double* soln, const int K, double* fitarray) {
        /* A function calculates the fitness values (reported in fitarray) of a
solution (soln) over a sample size of K.
        * This function simulates the adaptive phase interferometry and so is the
most computationally expensive part of the program.
        */
        dcmplx sharp(0.0, 0.0); //variable to store the sharpness function, which
is the first fitness value in the array.
        double error = 0.0; //variable to store the bias of the estimate, which is
the second fitness value in the array.
        bool dect_result; //variable to store which path a photon comes out at any
step.
        double PHI, phi, coin, PHI_in;
        int m, k, d;

        WK_state(); //Generate the WK state.

        for (k = 0; k < K; ++k) {
                phi = uniform_rng->next_rand(0.0, 1.0) * (upper - lower) + lower;
                PHI = 0;
                //copy input state: the optimal solution across all compilers is
memcpy:

//nadeausoftware.com/articles/2012/05/c_c_tip_how_copy_memory_quickly
                memcpy(state, input_state, (num + 1) * sizeof(dcmplx));
                //Begining the measurement of one sample
                d = 0;
                for (m = 0; m < num; ++m) {
                        // This loop is the most critical part of the entire
program. It
                        // executes K*num=10*num^3 times on each call of
avg_fitness. All
                        // optimization should focus on this loop.

                        //randomly decide whether loss occurs
                        coin = uniform_rng->next_rand(0.0, 1.0);

                        if (coin <= loss) {
                                //////////
                                state_loss(num - m); //update only the state using
loss function
                        }
                        else {
                                //PHI_in = rand_Gaussian(PHI, THETA_DEV); //select
if the noise is normally distributed.
                                //PHI_in = rand_Hat(PHI, THETA_DEV); //select if
the noise is distributed as a Hat function.
                                //PHI_in = Lognormal(MU,THETA,PHI); //select if
the noise is distributed as a lognormal function.
                                //PHI_in = rand_RTN(PHI,Ps,THETA_DEV);//select if
the noise is random telegraph.
                                PHI_in = rand_skewed(PHI, THETA_DEV, RATIO);
//select if the noise is skewed normal.
                                PHI_in = mod_2PI(PHI_in);//noisy PHI
                                dect_result = noise_outcome(phi, PHI_in, num - m);
                                if (dect_result == 0) {
                                        PHI = PHI - soln[d++];
                                }
                                else {
                                        PHI = PHI + soln[d++];
                                }
                                PHI = mod_2PI(PHI);
                        }
                }
                //store fitness values
                sharp.real(sharp.real() + cos(phi - PHI));
                sharp.imag(sharp.imag() + sin(phi - PHI));
        }
        //find the averages and return
        fitarray[0] = abs(sharp) / double(K); //Calculate the dispersion
        fitarray[1] = atan2(sharp.imag(), sharp.real()); //Calculate the mean
direction
}

bool Phase::T_condition(double* fitarray, int* numvar, int N_cut, bool* mem_ptype,
double* memory_forT) {
        /*This function contains the conditions that has to be checked after a
step T elapses
        * before the algorithm decides to accept or reject the solution.
        * In particular this function is called when time step is used as the main
```

```
condition to end the optimization.
        * For this particular problem, it resets the number of variables so the
optimization starts over.
        */
        //Let's start this condition from scratch.
        bool type;

        //The conditions are checked only if the algorithm is going to change the
way it initializes the population.
        //Policy type 1 is the one with pi bias and is susceptible to loss. The
algorithm will run until type 0 is found.
        if (*numvar == N_cut - 1) {
                try {
                        type = check_policy(fitarray[1], fitarray[0]);
                }
                catch (invalid_argument) {
                        fitarray[0] = 0.999999;
                        type = check_policy(fitarray[1], fitarray[0]);
                }
                mem_ptype[0] = type;
                if (type == 1) {
                        //*numvar = *numvar - 1;
                }
                else if (type == 0) {
                        memory_forT[0] = fitarray[0];
                        memory_forT[1] = fitarray[1];
                }
                else {}
        }
        else if (*numvar == N_cut) {
                try {
                        type = check_policy(fitarray[1], fitarray[0]);
                }
                catch (invalid_argument) {
                        fitarray[0] = 0.999999;
                        type = check_policy(fitarray[1], fitarray[0]);
                }
                mem_ptype[1] = type;
                //        if(mem_ptype[0] | type) {
                if (check_policy(memory_forT[1], memory_forT[0]) | type) {
                        //the policy is bad
                        //reset the policy found in numvar=N_cut-1
                        //*numvar = N_cut - 2;
                }
        }
        return 1;
}

bool Phase::error_condition(double* current_fitarray, double* memory_fitarray, int
data_size, double goal) {
        /*This function contains the function to compute error for when the
algorithm is set to use error as the accept-reject condition.
        *It allows for the information of previous optimization to be used to
compute the condition as well as using the latest result.
```

```
        *In this particular problem, we use the error that corresponds to the
confidence interval of 0.98
        * from the linear relation between logN and logV_H.
        */

        double slope, intercept;
        double mean_x;
        double tn2;
        double error, error_goal;

        int xy_data_size = data_size + 1;

        double* x = new double[xy_data_size];
        double* y = new double[xy_data_size];

        bool out;

        memory_fitarray[2 * (data_size)] = log10(num);
        memory_fitarray[2 * (data_size)+1] = log10(pow(current_fitarray[0], -2) -
1);

        //split into x-y arrays

        for (int i = 0; i <= data_size; ++i) {
                x[i] = memory_fitarray[2 * i];
                y[i] = memory_fitarray[2 * i + 1];
        }

        //Computing the linear equation using previous data
        linear_fit(data_size, x, y, &slope, &intercept, &mean_x);
        //Compute the goal for error from confidence interval of 0.98
        error_goal = error_interval(x, y, mean_x, data_size + 1, slope,
intercept);

        goal = (goal + 1) / 2;
        tn2 = quantile(goal);
        error_goal = error_goal * tn2;

        //Compute the distance between the data and the prediction from linear
equation
        error = y[data_size] - x[data_size] * slope - intercept;

        //Check if error is smaller than the goal
        if (error <= error_goal) {
                out = 1;
        }
        else {
                out = 0;
        }
        return out;
}

void Phase::boundary(double* can1) {
        /*This function cuts the value of the variables so they are within the
```

```
boundary of the problem.
        *This can be replaced by one or more method, such as periodic boundary or
normalization.
        */
        for (int i = 0; i < num; ++i) {
                if (can1[i] < lower_bound[i]) {
                        can1[i] = lower_bound[i];
                }
                else if (can1[i] > upper_bound[i]) {
                        can1[i] = upper_bound[i];
                }
        }
}
/*private functions: specific to the simulation and not used by the optimization
algorithm*/
/*########### Generating Input State ###########*/
/*Generation function*/
void Phase::WK_state() {
        /* This is the main function for generating the input state.
        * We call it here the Wiseman-Killip state, but the more general term
would be the sine state.
        * This algorithm for generating WK state is the most precise one to date
(measured by how much the normalization differs from zero),
        * but the rounding error still appears as early as N=80 and is detrimental
to the shape of the state for N>100.
        * This is a limitation caused by the use of double precision float.
        * For N>100, either switch to quadruple precision, which will cause a
large overhead
        * (about 15 for the simulation, but even more for this function), or
another method has to be used to approximate the state.
        */
        const double beta = M_PI / 2;
        const double cosN = pow(cos(beta / 2), num);
        tan_beta = tan(beta / 2);

        int n, k;
        double* n_part = new double[num + 1];
        double* k_part = new double[num + 1];
        double s_part;
        double temp;

        //Preparing constant arrays to save time.

        //initializing array of factorial numbers
        sqrtfac(sqrtfac_mat);
        one_over_fac(overfac_mat);

        /*factors in d_matrix calculated*/
        //calculating n_parts and k_parts in
        for (n = 0; n <= num; ++n) {
                n_part[n] = pow(-1.0, n) * sqrtfac_mat[n] * sqrtfac_mat[num - n] *
cosN * pow(tan_beta, n);
        }
        for (k = 0; k <= num; ++k) {
                k_part[k] = 1 / pow(-1.0, k) * sqrtfac_mat[k] * sqrtfac_mat[num -
k] * pow(1 / tan_beta, k);
        }

        //Compute the state
        for (n = 0; n <= num; ++n) { //we have N+1 state b/c we include n=0 and
n=N.
                temp = 0;
                input_state[n].real(0);
                input_state[n].imag(0);
                for (k = 0; k <= num; ++k) {
                        s_part = cal_spart(n, k, num);
                        temp = s_part * k_part[k] * sin((k + 1) * M_PI / (num +
2));
                        input_state[n].real(input_state[n].real() + temp *
cos(M_PI / 2.0 * (k - n)));
                        input_state[n].imag(input_state[n].imag() + temp *
sin(M_PI / 2.0 * (k - n)));
                }//end k
                input_state[n].real(input_state[n].real() * n_part[n] / sqrt(num /
2.0 + 1));
                input_state[n].imag(input_state[n].imag() * n_part[n] / sqrt(num /
2.0 + 1));
        }//end n

        delete[] n_part;
        delete[] k_part;

}

inline double Phase::cal_spart(const int n, const int k, const int N) {
        /*This function calculates the sum in an element of a Wigner d-matrix
        */

        int s;
        int s_min;
        int s_max;
        double s_part = 0;

        //find lower limit
        if (n - k >= 0) {
                s_min = 0;
        }
        else if (n - k < 0) {
                s_min = k - n;
        }
        else {}
        //find upper limit
        if (k <= N - n) {
                s_max = k;
        }
        else if (k > N - n) {
                s_max = N - n;
        }
        else {}

        //calculating the s_part
        for (s = s_min; s <= s_max; ++s) {
                s_part = s_part + pow(-1.0, s) * overfac_mat[k - s] *
overfac_mat[s] * overfac_mat[n - k + s] * overfac_mat[N - n - s] * pow(tan_beta, 2
* s);
        }
        return s_part;
}

inline void Phase::one_over_fac(double* over_mat) {
        /*This function calculates one over factorial matrix.*/
        over_mat[0] = 1;
        for (int i = 1; i <= num; ++i) {
                over_mat[i] = over_mat[i - 1] / i;
        }//end i
}

inline void Phase::sqrtfac(double* fac_mat) {
        /*This function calculates sqrare root of factorial matrix.*/
        //check array size
        fac_mat[0] = 1;
        for (int i = 1; i <= num; ++i) {
                fac_mat[i] = fac_mat[i - 1] * sqrt_cache[i];
        }//end i
}

/*########### Measurement Functions ###########*/
/*The following functions are involved in the simulation of the noisy
interferometer.*/

inline bool Phase::noise_outcome(const double phi, const double PHI, const int N)
{
        /*This function computes the output path of a photon from a noisy
interferometer
        by computing the probablity of photon coming out of either path.
        This simulation allows for noise in the unitary operation, but we only
consider the noise in the phase shift.
        */
        //N is the number of photons currently available, not equal to 'num'
        const double theta = (phi - PHI) / 2.0;
        const double cos_theta = cos(theta) / sqrt_cache[N];
        const double sin_theta = sin(theta) / sqrt_cache[N];
        //noise in operator: currently not in use
        const double oper_n0 = gaussian_rng->next_rand(0.0, DEV_N);//n_x
        const double oper_n2 = gaussian_rng->next_rand(0.0, DEV_N);//n_z
        const double oper_n1 = sqrt(1.0 - (oper_n0 * oper_n0 + oper_n2 *
oper_n2));
        const dcmplx U00(sin_theta * oper_n1, -oper_n0 * sin_theta);
        const dcmplx U01(cos_theta, oper_n2 * sin_theta);
        const dcmplx U10(cos_theta, -oper_n2 * sin_theta);
        const dcmplx U11(sin_theta * oper_n1, sin_theta * oper_n0);
        int n;
        double prob = 0.0;
        for (n = 0; n < N; ++n) {
                //if C_0 is measured
                update0[n] = state[n + 1] * U00 * sqrt_cache[n + 1] + state[n] *
U01 * sqrt_cache[N - n];
                prob += abs(update0[n] * conj(update0[n]));
        }

        if (uniform_rng->next_rand(0.0, 1.0) <= prob) {
                //measurement outcome is 0
                state[N] = 0;
                prob = 1.0 / sqrt(prob);
                for (n = 0; n < N; ++n) {
                        state[n] = update0[n] * prob;
                }
                return 0;
        }
        else {
                //measurement outcome is 1
                prob = 0;
                for (n = 0; n < N; ++n) {
                        state[n] = state[n + 1] * U10 * sqrt_cache[n + 1] -
state[n] * U11 * sqrt_cache[N - n];
                        prob += abs(state[n] * conj(state[n]));
                }
                state[N] = 0;
                prob = 1.0 / sqrt(prob);
                for (n = 0; n < N; ++n) {
                        state[n] *= prob;
                }
                return 1;
        }
}

inline void Phase::state_loss(const int N) {
        /*This function updates the state when one of the photon is loss.*/
        double total = 0;
        double factor = 1 / sqrt(2 * N);
        for (int n = 0; n < N; ++n) {
                state[n] = (state[n] * sqrt_cache[N - n] + state[n + 1] *
sqrt_cache[n + 1]) * factor;
                total += state[n].real() * state[n].real() + state[n].imag() *
state[n].imag();
        }
        state[N] = 0;
        //Necessary state renormalization
        for (int n = 0; n < N; ++n) {
                state[n] = state[n] / sqrt(total);
        }
}

inline double Phase::mod_2PI(double PHI) {
        /*This function compute the modulo of the phase.
        */
        while (PHI >= 2 * M_PI) {
                PHI = PHI - 2 * M_PI;
        }
        while (PHI < 0) {
                PHI = PHI + 2 * M_PI;
        }
        return PHI;
}

inline bool Phase::check_policy(double error, double sharp) {
        /*This function takes the bias and output the policy type.
        *A policy is considered to be type zero if its error falls within the
uncertainty of the scheme.
        *This is the desirable type as its estimate no bias and the policy can be
used when loss is presence.
        *In the error falls outside the uncertainty,
        *it is very likely that the estimate has a pi bias and is the type of
policy that fails when there is loss.
        */
        if (sharp == 1.0) {
                throw invalid_argument("sharpness cannot be one.");
        }
        double sd = sqrt(1 / (sharp * sharp) - 1);
        if (fabs(error) >= fabs(M_PI - sd)) {
                return 1;
        }
        else {
                return 0;
        }
}

double Phase::rand_Gaussian(double mean, /*the average theta*/
        double dev /*deviation for distribution*/
) {
        /*creating random number using Box-Muller Method/Transformation*/
        double Z0;//,Z1;
        double U1, U2; /*uniformly distributed random number input*/
        double r;

        /*create input between [-1,1]*/
        do {
                U1 = 2.0 * double(rand()) / RAND_MAX - 1.0;
                U2 = 2.0 * double(rand()) / RAND_MAX - 1.0;
                r = U1 * U1 + U2 * U2;
        } while (r == 0.0 || r >= 1.0);
        /*using Box-Muller Transformation*/
        Z0 = U1 * sqrt(-2 * log(r) / r);

        return Z0 * dev + mean;
}/*end of rand_Gaussian*/

double Phase::rand_Hat(double PHI, double dev) {
        return gaussian_rng->next_rand(PHI, dev);
}
```

```
inline double Phase::inv_erf(double x) {
        if (x == 1) {
                throw invalid_argument("Input leads to error.");
        }
        double a = 0.140012;
        double lnx = log(1 - x * x);
        double temp = sqrt(pow(2.0 / (M_PI * a) + lnx / 2.0, 2) - lnx / a);

        return sgn(x) * sqrt(temp - 2.0 / (M_PI * a) - lnx / 2.0);
}

inline int Phase::sgn(double x) {
        /** Sign function of x: https://en.wikipedia.org/wiki/Sign_function
        */
        if (x < 0) {
                return -1;
        }
        else if (x == 0) {
                return 0;
        }
        else {
                return 1;
        }
}

double Phase::Lognormal(double mu, double sigma, double peak) {
        double mode = exp(mu - sigma * sigma); //where the peak is, so we know how
much to move the distribution later.
        double diff = mode - peak;
        double ans;

        double u = (double)(rand()) / ((double)(RAND_MAX));
        double p = 2 * u - 1;

        try {
                ans = exp(sqrt(2) * sigma * inv_erf(p) + mu) - diff;
        }
        catch (invalid_argument) {
                u = (double)(rand()) / ((double)(RAND_MAX));
                p = 2 * u - 1;
                ans = exp(sqrt(2) * sigma * inv_erf(p) + mu) - diff;
        }

        return ans;
}

double Phase::rand_RTN(double PHI, double ps, double dev) {
        double coin = double(rand()) / double(RAND_MAX);
        double ans;
        if (coin <= ps) {
                coin = double(rand()) / double(RAND_MAX);
                if (coin < 0.5) {
                        ans = PHI + dev;
                }
                else {
                        ans = PHI - dev;
                }
        }
        else {
                ans = PHI;
        }
        return ans;
}

double Phase::rand_skewed(double mean, double dev, double ratio) {
        double u1, u2;
        double k1, k2;

        double alpha = ratio * dev;

        k1 = (1 + alpha) / sqrt(2 * (1 + alpha * alpha));
        k2 = (1 - alpha) / sqrt(2 * (1 + alpha * alpha));

        u1 = rand_Gaussian(0, 1);
        u2 = rand_Gaussian(0, 1);

        return (k1 * max(u1, u2) + k2 * min(u1, u2)) * dev + mean;
}

#endif // PHASE_H
```

problems.h

```
#ifndef PROBLEM_H
#define PROBLEM_H

using namespace std;

/*! \brief Problem class contains the prototype of the functions in the
optimization problem that OptAlg class needs.
*/

class Problem
{
public:
        Problem() {};
        virtual ~Problem() {
                delete[] upper_bound;
                delete[] lower_bound;
        }

        virtual void fitness(double* soln, double* fitarray) {
                /*! A function intend to be a wrapper for changing conditions in
which the fitness function is evaluated.*/
        }
        virtual void avg_fitness(double* soln, int K, double* fitarray) {
                /*! A function for calculating the fitness value.
                It allows a number of sample K to be passed into the function in
case the fitness function is a statistical 'quantity'(?)*/
        }
        virtual bool T_condition(double* fitarray, int* numvar, int N_cut, bool*
mem_ptype, double* memory_forT) {
                /*! A function for calculating additional conditions for when the
optimization algorithm is set to accept solution after time T.*/
                return 0;
        }
        virtual bool error_condition(double* current_fitarray, double*
memory_fitarray, int data_size, double goal) {
                /*! A function for calculating additional conditions for when
optimization algorithm is set to accept solution from error bound.*/
                return 0;
        }
        virtual void boundary(double* can1) {
                /*! This function is used to keep the solution candidate within
the boundary of the search space.*/
        }

        double* lower_bound;/*!< Pointer to array storing the lower bound of the
variables*/
        double* upper_bound;/*!< Pointer to array storing the upper bound of the
variables*/
```

```
        int num; /*!<number of variables in the problem*/
        int num_repeat; /*!<number of repeats*/
        int num_fit; /*<number of fitnesses*/
};

#endif // PROBLEM_H
```

rng_vsl.cpp

```
#include <iostream>
#include <map>
#include <vector>
#include "mkl_vsl.h"

#include "rng.h"

#define BRNG VSL_BRNG_MCG31
#define METHOD VSL_RNG_METHOD_GAUSSIAN_ICDF

VSLStreamStatePtr stream;

using namespace std;

RngVsl::RngVsl(bool _gaussian, int _n_random_numbers, int seed, int rank):
    n_random_numbers(_n_random_numbers), RngVectorized(_gaussian) {
    vslNewStream(&stream, BRNG, seed + rank);
    random_numbers = new double[n_random_numbers];
    index_random_numbers = 0;
    if (gaussian) {
        vdRngGaussian(METHOD, stream, n_random_numbers, random_numbers, 0.0, 1.0);
        }
    else {
        vdRngUniform(VSL_RNG_METHOD_UNIFORM_STD, stream, n_random_numbers,
random_numbers, 0.0, 1.0);
        }
    }

double RngVsl::next_rand(const double mean, const double dev) {
    if (index_random_numbers >= n_random_numbers) {
        index_random_numbers = 0;
        if (gaussian) {
            vdRngGaussian(METHOD, stream, n_random_numbers, random_numbers, 0.0,
1.0);
            }
        else {
            vdRngUniform(VSL_RNG_METHOD_UNIFORM_STD, stream, n_random_numbers,
random_numbers, 0.0, 1.0);
            }
        }
    return random_numbers[index_random_numbers++] * dev + mean;
    }

RngVsl::~RngVsl() {
    delete[] random_numbers;
    vslDeleteStream(&stream);
    }
```

rng.cpp

```
#include <iostream>
#include <cstdlib>
#include <cstring>

#include "rng.h"

RngSimple::RngSimple(bool _gaussian, int n_random_numbers, int seed, int rank):
    RngBase(_gaussian) {
    srand(seed + rank);
    }

RngSimple::~RngSimple() {
    }

double RngSimple::next_rand(const double mean, const double dev) {
    if (gaussian) {
        return (double(rand()) / RAND_MAX - 0.5) * 2 * dev + mean;
        }
    else {
        return double(rand()) / RAND_MAX;
        }
    }
```

rng.h

```
#ifndef RNG_H
#define RNG_H
#if HAVE_CONFIG_H
#include <config.h>
#endif
#ifdef CUDA
#include <curand.h>
#endif

using namespace std;

/*!  \brief Rng class store methods for generating random numbers using RngBase as
its base class.
*    The class can generate uniformaly random number for approximating normally
distributed numbers when 'gaussian' is specified.
*/

class RngBase {
    public:
        RngBase(bool _gaussian): gaussian(_gaussian) {};
        ~RngBase() {};
        double next_rand(const double mean, const double dev) {
            return 0.0;
            };

    protected:
        bool gaussian;
    };

/*!   \brief RngSimple generates random numbers from C++ build-in pseudo-random
number generator.
*/

class RngSimple: public RngBase {
    public:
        RngSimple(bool _gaussian, int n_random_numbers, int seed, int rank);
        ~RngSimple();
        double next_rand(const double mean, const double dev);
    };

/*! \brief RngVectorized generates random numbers into vectors in order to reduce
the computational overhead.
*   There are two methods to this class: VSL and GPU.
*/

class RngVectorized: public RngBase {
    public:
        RngVectorized(bool _gaussian): RngBase(_gaussian) {};
        ~RngVectorized() {};
        double next_rand(const double mean, const double dev);
    };

#ifdef CUDA
/*! \brief RngGpu generates vectors of random number asynchronously on GPUs. One
GPU is assigned per CPU.
*/

class RngGpu: public RngVectorized {
    public:
        RngGpu(bool _gaussian, int n_random_numbers, int seed, int rank);
        ~RngGpu();
        double next_rand(const double mean, const double dev);

    private:
        double *random_numbers;
        int n_random_numbers;
        int index_random_numbers;

        double *dev_random_numbers;
        curandGenerator_t gen;
    };

#define Rng RngGpu

#elif defined(VSL)

/*!    \brief RngVsl uses Intel's VSL library to generate a vector of random
numbers. This approach reduce computational overhead.
*/

class RngVsl: public RngVectorized {
    public:
        RngVsl(bool _gaussian, int n_random_numbers, int seed, int rank);
        ~RngVsl();
        double next_rand(const double mean, const double dev);

    private:
        double *random_numbers;
        int n_random_numbers;
        int index_random_numbers;
    };
#define Rng RngVsl

#else

#define Rng RngSimple

#endif

#endif // RNG_H
```