Homework 7

Problem

1. Implement Genetic algorithm to seek the global minimum of the following functions:
   (a) $f(x) = 2(x - 0.5)^2 + 1$ on $[0, 1]$
   (b) $f(x) = |x - 0.5|(\cos(12\pi[x - 0.5]) + 1.2)$ on $[0, 1]$
2. Discuss its performance depending on various control parameters such as length of genotype, population size, mutation probability, crossover probability and so on.

Implementation

1. **Chromosome & Population**
   (a) Due to the implementation convienience, *chromosome_t* and *population_t* are predefined. *chromosome_t* is a encoded binary type and *population_t* contains a *chromosome_t* and their fitness value.
   (b) It is more complicate to convert floating point value to binary value than to convert integer value. I used a trick here. As shown in the code, there is no encoding function converting from floating point to binary, there is just a decoding function. The binary value of *chromosome_t* is automatically generated by bernoulli distribution. After then, when it is decoded, the chromosome value becomes normalized into range 0 to 1 by the value of a maximum bitstring.

```cpp
#ifndef __GA_HELPER__
#define __GA_HELPER__

#include "method.h"

#include <random>
#include <bitset>
#include <fstream>

namespace numerical_optimization {

template<typename pheno_t, size_t size>
struct chromosome_t {
    using bit_t = std::bitset<size>;

    chromosome_t(){
        // binary distribution
        std::bernoulli_distribution d(0.5);
        for(size_t i=0; i<size; i++)
```

```cpp
            gene[i] = d(gen);
    };

    inline pheno_t decode() {
        return pheno_t(gene.to_ullong())/pheno_t(bit_t(ULLONG_MAX).to_ullong());
    }

    void mutate() {
        std::uniform_int_distribution<size_t> dis(0, size-1);
        size_t index = dis(gen);

        gene.flip(index);
    }

    void crossover(chromosome_t<pheno_t, size>& other, size_t end, size_t start=0) {

        size_t min, max;
        std::tie(min, max) = std::minmax(end, start);

        // one point & two points crossover
        bit_t t_this  = this->gene;
        bit_t t_other = other.gene;
        for(size_t i=max; i>min; i--) {
            this->gene[i] = t_other[i];
            other.gene[i] = t_this[i];
        }
    }

    const std::string to_string() const {
        return gene.to_string();
    }
private:
    bit_t gene;
};

template<typename vector_t, typename scalar_t, size_t size>
struct population_t {

    using function_t = std::function<scalar_t(const vector_t&)>;

    population_t():chromosome(chromosome_t<vector_t, size>()),fitness(0),probability(0)
↪   {}

    scalar_t eval_fitness(const function_t& function){
        return fitness = function(chromosome.decode());
    };

    double eval_probability(const scalar_t& sum_fitness, const size_t& total_size) {
        return probability = ((sum_fitness-fitness)/sum_fitness)/(total_size-1);
    }

    chromosome_t<vector_t, size> chromosome;
    scalar_t fitness;
    double   probability;
};
//// ////////////////////////////////////////////////
}/// the end of namespace numerical_optimization ///
////////////////////////////////////////////////////
#endif //__GA_HELPER__
```

2. **GeneticAlgorithm**

   (a) Initialization

   - The popuations are initialized as given size by population_t and their fitness values are also evaluated. And all this processes are done in the construction of *GeneticAlgorithm*.

   (b) Selection

   - I implemented roulette wheel selection as selection method
   - Because we want to find minimum value of the given function, the probabilty to choose chromosome is computed reversely from the fitness values
     $prob = (1 - fitness/fitness\_sum)/thenumberofchromosomes$

   (c) Reproduction

   - As a reproduction, crossover and mutation can occur to the selected chromosomes within the given probability. The performance related to this will be discussed later.
   - crossover - point slicing method using twopoint, it is determined by uniform distribution in the chromosome length

   (d) Replacement

   - As the replacement policy, there are the two methods.
   - One of these is *replace_parent*, which replace the selected chromosomes to the new chromosomes that are crossovered or mutated.
   - The other is *replace_worst*. By searching the worst fitness value within the populations, we can replace the chromosome which has the worst fitness value to a new chromosome generated by crossover or mutation method.
   - In the performance comparison part, I fixed the replacement method as *replace_worst*

```cpp
#ifndef __GENETIC_ALGORITHM__
#define __GENETIC_ALGORITHM__

#include "method.h"
#include "ga_helper.h"

#include <random>
#include <bitset>
#include <fstream>

namespace numerical_optimization {

template<typename vector_t, typename scalar_t, size_t len>
class GeneticAlgorithm : Method {
```

```
public:
    using boundary_t = std::pair<scalar_t, scalar_t>;
    using function_t = std::function<scalar_t(const vector_t&)>;
    using populate_t = population_t<vector_t, scalar_t, len>;

    GeneticAlgorithm(
        function_t func,
        size_t size=50,
        double crossover_prob=0.7,
        double mutation_prob=0.1
        )
    :function(func)
    ,population(size)
    ,prob_crossover(crossover_prob)
    ,prob_mutation(mutation_prob)
    ,sum_fitness(0)
    ,sum_probability(0) {
        initialize_population();
        evaluate_fitness();
    }

    // iteration run
    double run(size_t iteration) {

        for(size_t i=0; i<iteration; i++) {
            // selection
            size_t idx_x, idx_y;
            std::tie(idx_x, idx_y) = select_with_roulettewheeling();

            //////////////////////////////////////////////////
            // reproduction
            //////////////////////////////////////////////////
            populate_t x = population[idx_x];
            populate_t y = population[idx_y];
            // cross over
            if(crossover(x, y)) {
                x.eval_fitness(function);
                y.eval_fitness(function);
            }

            // mutation
            if(mutation(x) || mutation(y)) {
                x.eval_fitness(function);
                y.eval_fitness(function);
            }

            //////////////////////////////////////////////////
            // replacement
            //////////////////////////////////////////////////
            // replacement worst
```

```cpp
            {
                replace_worst(x);
                replace_worst(y);
            }

            // update whole populations; lazy method
            evaluate_fitness();
        }
        return sum_fitness/population.size();
    }

    void print(std::string filepath) {

        std::ofstream file(filepath.data());

        if(file.is_open()) {
            file << "chromosome ";
            file << "decoded ";
            file << "fitness\n";

            for(auto& p:population) {
                file << p.chromosome.to_string() << " ";
                file << p.chromosome.decode() << " ";
                file << p.fitness << std::endl;
            }
            file << "\n";
            file << "sum of fitness: " << sum_fitness/population.size() << "\n";
        }
        file.close();
    }

private:
    void initialize_population() {
        // randomly initialize population from chromosome
        for(size_t i=0; i<population.size(); i++) {
            population[i] = populate_t();
        }
    }

    void evaluate_fitness() {
        // evaluate fitness from
        sum_fitness = 0;
        for(auto& p:population)
            sum_fitness += (p.eval_fitness(function));

        sum_probability = 0;
        for(auto& p:population)
            // todo
            sum_probability += p.eval_probability(sum_fitness, population.size());
    }

    //==========================================================
    // selection algorithm
    //==========================================================
    // roulette wheeling selection
    std::tuple<size_t, size_t> select_with_rouletewheeling() {

        auto select = [&]() {
            std::uniform_real_distribution<scalar_t> dis(0.0, 1.0);
            scalar_t random_number = dis(gen);

            size_t i=0;
            while(random_number>0.0) {
                random_number -= population[i].probability;
                i++;
            }
            return i;
        };

        return std::make_tuple(select(), select());
    }

    //==========================================================
    // crossover algorithm
    //==========================================================
    bool crossover(populate_t& x, populate_t& y) {

        std::uniform_real_distribution<double> prob(0.0, 1.0);
        if(prob_crossover<prob(gen)) return false;

        std::uniform_int_distribution<size_t> range(0, len-1);
        size_t r_start = range(gen);
        size_t r_end   = range(gen);

        x.chromosome.crossover(y.chromosome, r_start, r_end);
        return true;
    }

    //==========================================================
    // mutation algorithm
    //==========================================================
    bool mutation(populate_t& x) {
        // convetional mutation
        std::uniform_real_distribution<double> prob(0.0, 1.0);
        if(prob_mutation<prob(gen)) return false;

        // mutate
        x.chromosome.mutate();
        return true;
    }
    //==========================================================
    // replacement algorithm
    //==========================================================
    // replace parent
    void replace_parent(size_t index, const populate_t x) {
        population[index] = x;
    }
    // replace worst: the highest fitness value
    void replace_worst(const populate_t x) {
        auto elem = std::max_element(population.begin(), population.end(), [](populate_t
 ↪  a, populate_t b){ return a.fitness<b.fitness; });
        (*elem) = x;
    }

protected:
    function_t  function;
    std::vector<populate_t> population;
    scalar_t    sum_fitness;
    double      sum_probability;
    double      prob_crossover;
    double      prob_mutation;
};

//// /////////////////////////////////////////////////
}/// the end of namespace numerical_optimization ///
////////////////////////////////////////////////////
#endif //__GENETIC_ALGORITHM__
```

Performance

1. Comparison

   - For the comparision of a performance, the normailized fitness value is used, which is the dividied sum of fitness value by population sizes. Also, this expericence is done under the manually given fixed number of iterations.
   - The one of records has been chosen and written into the tables, after multiple atttempting
   - As the result, the length of genotype has little impact on the global procedure. Howerver, I think that this factor would be related to the probability of mutation.
   - Otherwise, the larger population size shows the need for more iterations. That means a large population size increase the diversity of chromosome in the population. Moreover, the higher crossover probability and mutation probability make more diversity in the population, which means that these do the role as exploring (global) procedure.
   - length of genotype
     - population size: 50, crossover probability: 0.7, mutation probability: 0.1
     - As shown in the table, it looks like that the length of genotype little affect the convergence

function	iter	sum of fitness value			
		8	16	32	64
a	5	1.05291	1.06713	1.04837	1.05929
b	30	0.0702615	0.0396148	0.0710507	0.0303201

   - population size
     - length of genotype: 16, crossover probability: 0.7, mutation probability: 0.1
     - In the function a and b, population size is highly related to the convergence. When it has larger population size, it needs more iterations.
     - We can interpret this as more diversity in larger population size

function	iter	sum of fitness value			
		20	30	50	100
a	5	1.04197	1.05326	1.04957	1.07366
b	30	0.0161354	0.0378164	0.0597946	1.115687

   - crossover probability

  - population size: 50, length of genotype: 16, mutation probability: 0.1
  - In the function b case, the result shows apparently that exploring procedure have occured

function	iter	sum of fitness value			
		0.1	0.4	0.7	1.0
a	5	1.05877	1.07744	1.07088	1.04379
b	30	0.0598181	0.0595787	0.063038	0.0657014

- mutation probability
  - population size: 50, length of genotype: 16, crossover probability: 0.7
  - In the function b case, the result shows that exploring procedure have occured

function	iter	sum of fitness value			
		0.1	0.4	0.7	1.0
a	5	1.07561	1.04943	1.05938	1.05369
b	30	0.0406014	0.0322939	0.0479334	0.0510157