Title:
Comparative Analysis of Genetic Algorithms in Optimization Problems

Under the guidance of
Dr. Haider Banka
(Associate Professor)

Department of Computer Science and Engineering
INDIAN INSTITUTE OF TECHNOLOGY
(INDIAN SCHOOL OF MINES)
DHANBAD

By: Avinesh Pratap Singh
20JE0219

Index

SUBJECT - SOFT COMPUTING
SUBJECT CODE - CSO505
NAME - Avinesh Pratap Singh
ADMISSION No. - 20JE0219

Assignment Problem:

Write programs in C/C++ only. One using simple GA and one using NSGA II. Each one should run on two benchmark functions as assigned in each group. The report to be submitted with the following format. A suitable Title of report with Name & Admission No. on it. Problem statements, Working Principle of the algorithm in your own English, codes with input and outputs, hands-on calculation to illustrate the whole process/simulation for at least two iterations.

Problems Description:

Functions	Constraints	Search Domain
$\text{Minimize} = \begin{cases} f_1(\boldsymbol{x}) = \sum_{i=1}^{2}\left[-10\exp\left(-0.2\sqrt{x_i^2 + x_{i-1}^2}\right)\right] \\ f_2(\boldsymbol{x}) = \sum_{i=1}^{3}\left[\lvert x_i\rvert^{0.8} + 5\sin\left(x_i^3\right)\right] \end{cases}$		$-5 \leq x_i \leq 5$, $1 \leq i \leq 3$.
$\text{Minimize} = \begin{cases} f_1(x) = x^2 \\ f_2(x) = (x-2)^2 \end{cases}$		$-A \leq x \leq A$. Values of A from 10 to 10^5 have been used successfully. Higher values of A increase the difficulty of the problem.

1. Introduction

In this report, we explore the optimization capabilities of Genetic Algorithms (GA) and Non-dominated Sorting Genetic Algorithm II (NSGA-II) on benchmark functions. The primary objective is to compare the performance of these algorithms and analyse their efficiency in solving optimization problems.

2. Working principle of Genetic Algorithms (GA):

2.1. Initialization:

- Initialize a population of candidate solutions randomly or using specific strategies.
- Each candidate solution represents a potential solution to the optimization problem.

2.2. Fitness Evaluation:

- Evaluate the fitness of each candidate solution in the population.
- The fitness function quantifies how suitable each solution is based on problem-specific criteria.
- Higher fitness values indicate better solutions.

2.3. Selection:

- Use selection mechanisms such as roulette wheel selection, tournament selection, or rank-based selection to choose individuals for reproduction.
- Individuals with higher fitness have a higher chance of being selected, mimicking the concept of survival of the fittest.

2.4. Crossover (Recombination):

- Perform crossover (also known as recombination) to create new candidate solutions.

- Common crossover techniques include single-point crossover, multi-point crossover, and uniform crossover.
- Crossover combines genetic information from two parent solutions to produce offspring.

2.5. Mutation:

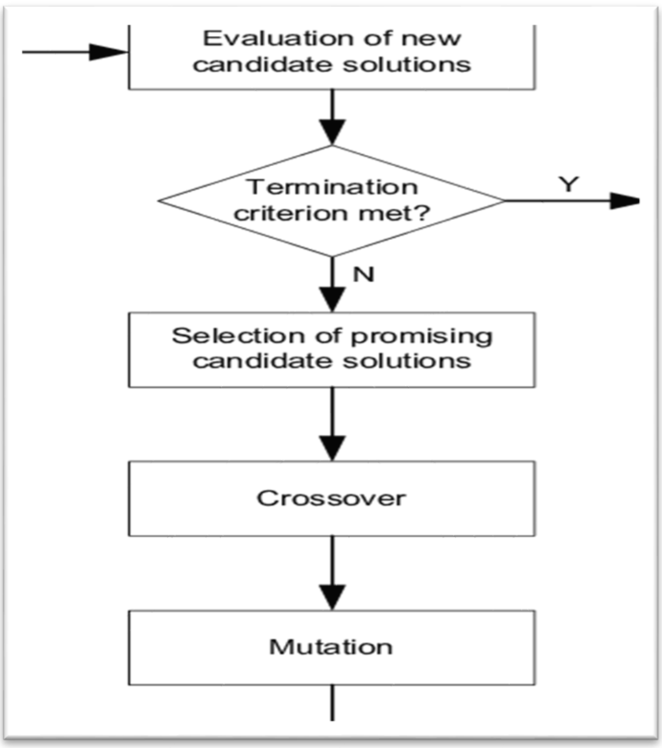
- Apply mutation to introduce small random changes in the offspring's genetic information.
- Mutation helps explore new areas of the solution space and prevents premature convergence to suboptimal solutions.

2.6. Replacement:

- Replace individuals in the current population with the offspring generated through crossover and mutation.
- Some strategies involve elitism, where the best-performing individuals from the current population are preserved.

2.7. Termination:

- Determine termination conditions such as reaching a maximum number of generations, achieving a satisfactory fitness level, or reaching a computational limit.
- Terminate the algorithm when the termination conditions are met.



3. Working principle of Non-dominated Sorting Genetic Algorithm II (NSGA-II):

3.1. Initialization:

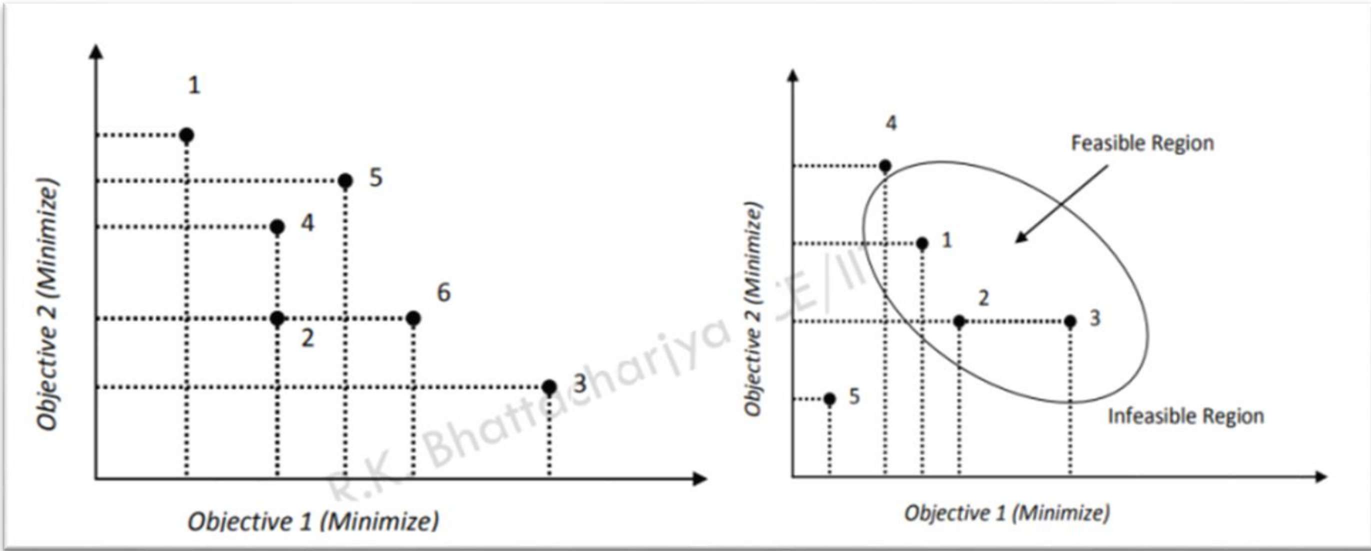
- Initialize a population of candidate solutions, similar to GA.

3.2. Fitness Evaluation:

- Evaluate the fitness of each candidate solution using multiple objective functions (for multi-objective optimization).
- NSGA-II deals with optimizing solutions that are non-dominated, i.e., no other solution in the population is better in all objectives and at least one objective is better.

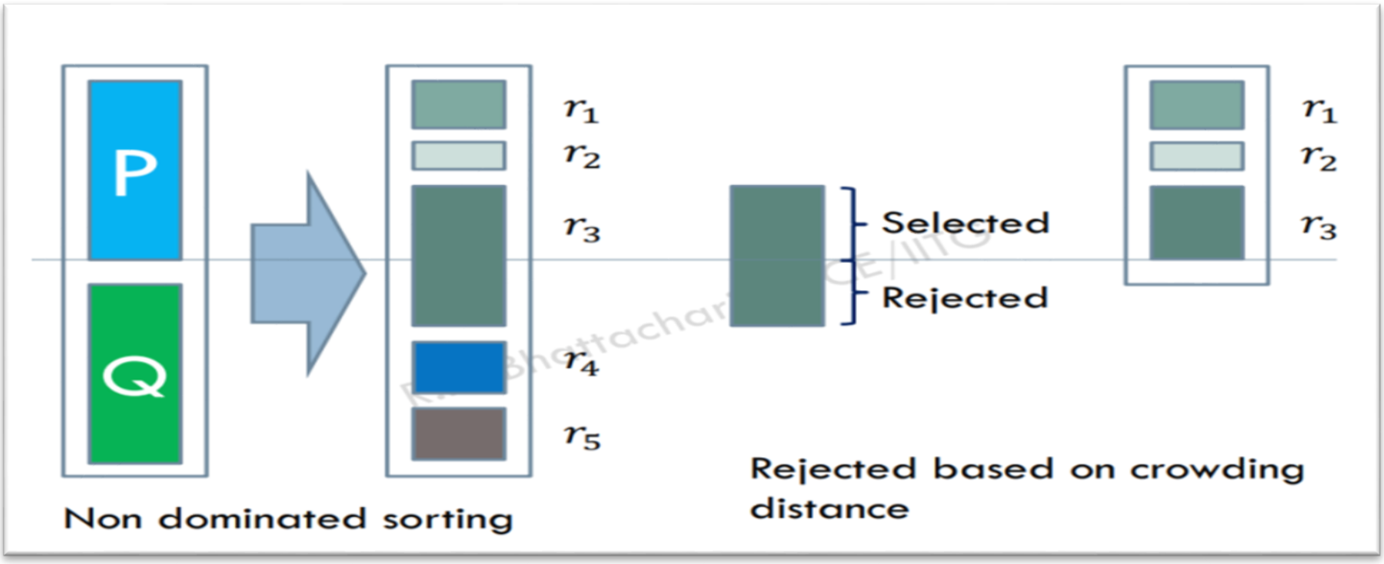
3.3. Non-dominated Sorting:

- Perform non-dominated sorting to categorize solutions into different fronts based on their dominance relationships.
- A solution is non-dominated if there is no other solution in the population that is better in all objectives.
- Solutions in the first front are non-dominated and represent the Pareto front.



3.4. Crowding Distance Calculation:

- Calculate the crowding distance for solutions in each front.
- Crowding distance measures the density of solutions in the objective space.
- Solutions with higher crowding distance are preferred to maintain diversity on the Pareto front.



3.5. Selection:

- Use a combination of non-dominated sorting and crowding distance to select individuals for reproduction.
- Solutions in less crowded regions of the Pareto front have a higher chance of being selected.

3.6. Crossover and Mutation:

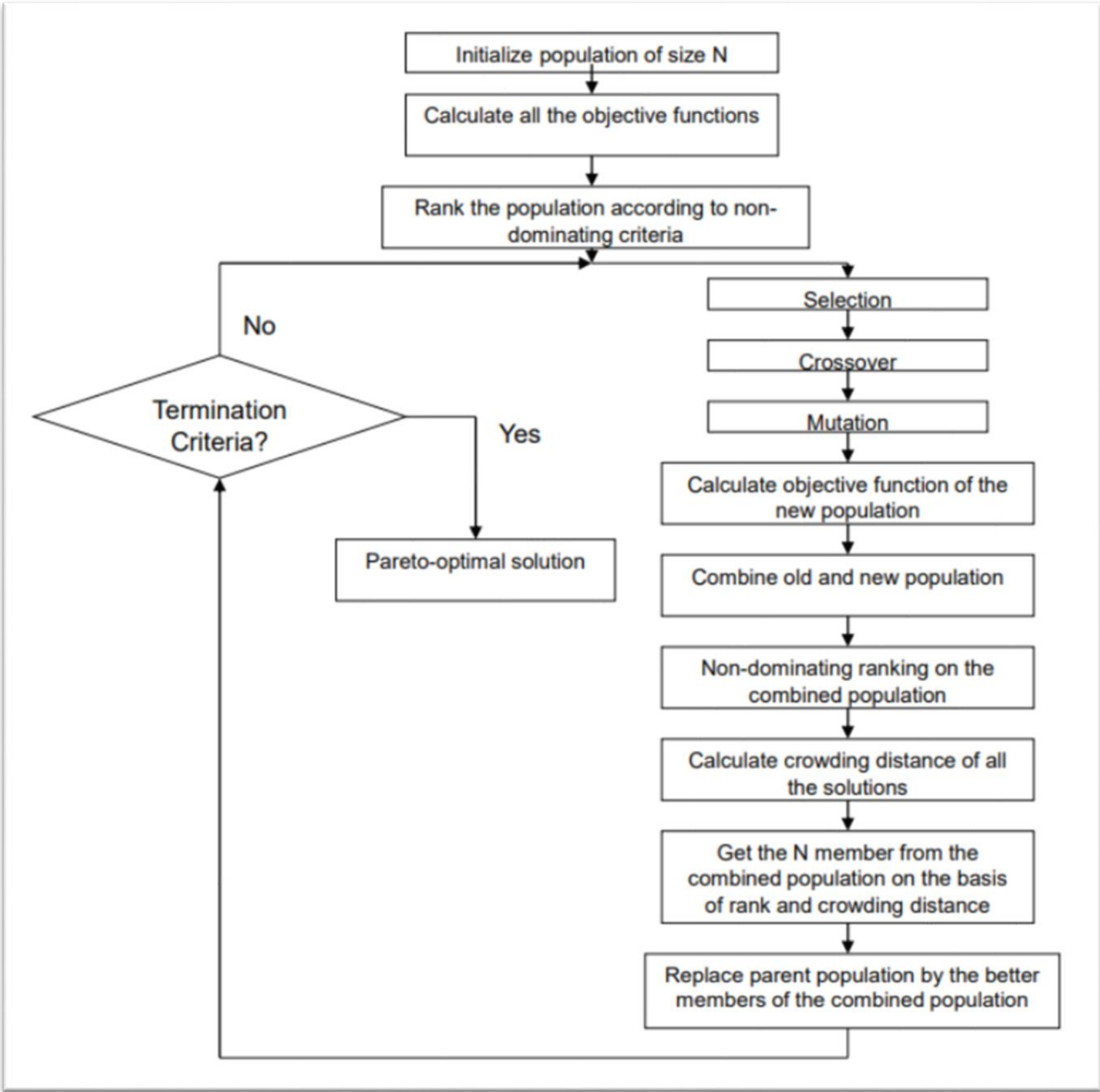
- Apply crossover and mutation operators to create new offspring solutions.
- Maintain diversity and balance between exploration and exploitation.

3.7. Replacement:

- Replace individuals in the current population with offspring based on selection criteria.
- Preserve non-dominated solutions and maintain diversity.

3.8. Termination:

- Terminate the algorithm based on termination conditions similar to GA.



4. Code:

The code for GA and NSGAII is single file and structured format project (different header have different work to do).
You can find the GitHub repository for 2^{nd} one here:
https://github.com/pratapavinesh/ga-and-nsgaii

For visualising graphs in both type of the project please download the gnuplot from here:
http://www.gnuplot.info/download.html

I have also given the separate code but you can change the problem as well in same code.
for changing the problem, please update the main function with corresponding problem.

Problem 5:

Functions	Constraints	Search Domain
$\text{Minimize} = \begin{cases} f_1(\boldsymbol{x}) = \sum_{i=1}^{2}\left[-10\exp\left(-0.2\sqrt{x_i^2 + x_{i-1}^2}\right)\right] \\ f_2(\boldsymbol{x}) = \sum_{i=1}^{3}\left[\lvert x_i\rvert^{0.8} + 5\sin\left(x_i^3\right)\right] \end{cases}$		$-5 \le x_i \le 5,$ $1 \le i \le 3.$

C++ code for the SGA:

```
#ifndef MATH_AUX__
#define MATH_AUX__

#include <cstdlib>
#include <vector>

namespace MathAux
{
	const double PI = 3.1415926;
	const double EPS = 1.0e-14; // follow nsga-ii source code
	inline double square(double n) { return n * n; }
	inline double random(double lb, double ub) { return lb + (static_cast<double>(std::rand()) / RAND_MAX) * (ub - lb); }

	// ASF(): achievement scalarization function
	double ASF(const std::vector<double>& objs, const std::vector<double>& weight);
}

#endif

#ifndef BASE_PROBLEM__
#define BASE_PROBLEM__

#include <string>
#include <vector>

// ----------------------------------------------------------------------
//                    BProblem: the base class of problems (e.g. ZDT and DTLZ)
// ----------------------------------------------------------------------
class CIndividual;

class BProblem
{
public:
	explicit BProblem(const std::string& name) :name_(name) {}
	virtual ~BProblem() {}

	virtual std::size_t num_variables() const = 0;
	virtual std::size_t num_objectives() const = 0;
	virtual bool Evaluate(CIndividual* indv) const = 0;

	const std::string& name() const { return name_; }
	const std::vector<double>& lower_bounds() const { return lbs_; }
	const std::vector<double>& upper_bounds() const { return ubs_; }
```

```
protected:
	std::string name_;

	std::vector<double> lbs_, // lower bounds of variables
		ubs_; // upper bounds of variables
};

#endif


#ifndef ASSIGNMENT_PROBLEM__
#define ASSIGNMENT_PROBLEM__

// #include "base_problem.h"
#include <cstddef>
#include<iostream>

// ---------------------------------------------------------------------
class CProblemAssignment : public BProblem
{
public:
	CProblemAssignment(std::size_t M, std::size_t k, const std::string& name, double lbs, double ubs);

	virtual std::size_t num_variables() const { return k_; }
	virtual std::size_t num_objectives() const { return M_; }

	virtual bool Evaluate(CIndividual* indv) const = 0;

protected:
	std::size_t M_; // number of objectives
	std::size_t k_; // number of variables
};


// ---------------------------------------------------------------------
//			CProblemAssignment5
// ---------------------------------------------------------------------

class CProblemAssignment5 : public CProblemAssignment
{
public:
	explicit CProblemAssignment5(std::size_t M, std::size_t k, double lbs, double ubs) :CProblemAssignment(M, k, "Assignment5", 
lbs, ubs) {}
	virtual bool Evaluate(CIndividual* indv) const;
};


// ---------------------------------------------------------------------
//			CProblemAssignment6
// ---------------------------------------------------------------------

class CProblemAssignment6 : public CProblemAssignment
{
public:
	explicit CProblemAssignment6(std::size_t M, std::size_t k, double lbs, double ubs) :CProblemAssignment(M, k, "Assignment6", 
lbs, ubs) {
		//std::cout << "what happend !" << lbs << '\n';
	}
	virtual bool Evaluate(CIndividual* indv) const;
};

#endif
```

```
#ifndef COMPARATOR__
#define COMPARATOR__

class CIndividual;

// ----------------------------------------------------------------------------------
//                                  BComparator : the base class of comparison operators
// ----------------------------------------------------------------------------------
class BComparator
{
public:
	virtual ~BComparator() {}

	virtual bool		operator()(const CIndividual& l, const CIndividual& r) const = 0;
};



// ----------------------------------------------------------------------------------
//                                  CParetoDominate
// ----------------------------------------------------------------------------------

class CParetoDominate : public BComparator
{
public:
	virtual	bool	operator()(const CIndividual& l, const CIndividual& r) const;
};


extern CParetoDominate ParetoDominate;
#endif


#ifndef CROSSOVER__
#define CROSSOVER__

// ----------------------------------------------------------------------------------
//                         CSimulatedBinaryCrossover : simulated binary crossover (SBX)
// ----------------------------------------------------------------------------------


class CIndividual;
class CSimulatedBinaryCrossover
{
public:
	explicit CSimulatedBinaryCrossover(double cr = 1.0, double eta = 30) :cr_(cr), eta_(eta) {} // NSGA-III (t-EC 2014) setting

	void SetCrossoverRate(double cr) { cr_ = cr; }
	double CrossoverRate() const { return cr_; }
	void SetDistributionIndex(double eta) { eta_ = eta; }
	double DistributionIndex() const { return eta_; }

	bool operator()(CIndividual* c1, CIndividual* c2, const CIndividual& p1, const CIndividual& p2, double cr, double eta) const;
	bool operator()(CIndividual* c1, CIndividual* c2, const CIndividual& p1, const CIndividual& p2) const
	{
		return operator()(c1, c2, p1, p2, cr_, eta_);
	}

private:

	double get_betaq(double rand, double alpha, double eta) const;

	double cr_; // crossover rate
```

```
	double eta_; // distribution index
};


#endif

#include <string>
#include <stdio.h>

#include <string>
#include <cstddef>

// ---------------------------------------------------------------------
// Gnuplot
//
// This is just a very simple interface to call gnuplot in the program.
// Now it seems to work only under windows + visual studio.
// ---------------------------------------------------------------------

class Gnuplot
{
public:
	Gnuplot();
	~Gnuplot();

	// prohibit copying (VS2012 does not support 'delete')
	Gnuplot(const Gnuplot&);
	Gnuplot& operator=(const Gnuplot&);

	// send any command to gnuplot
	void operator ()(const std::string& command);

	void reset() { operator()("reset"); }
	void replot() { operator()("replot"); }
	void set_title(const std::string& title);

	void plot(const std::string& fname, std::size_t x, std::size_t y);
	void splot(const std::string& fname, std::size_t x, std::size_t y, std::size_t z);

protected:
	FILE* gnuplotpipe;
};

#ifndef INDIVIDUAL__
#define INDIVIDUAL__

#include <vector>
#include <ostream>

// ---------------------------------------------------------------------
//			CIndividual
// ---------------------------------------------------------------------

class BProblem;

class CIndividual
{
public:
	typedef double TGene;
	typedef std::vector<TGene> TDecVec;
	typedef std::vector<double> TObjVec;

	explicit CIndividual(std::size_t num_vars = 0, std::size_t num_objs = 0);
```

```
    TDecVec& vars() { return variables_; }
    const TDecVec& vars() const { return variables_; }

    TObjVec& objs() { return objectives_; }
    const TObjVec& objs() const { return objectives_; }

    TObjVec& conv_objs() { return converted_objectives_; }
    const TObjVec& conv_objs() const { return converted_objectives_; }

    // if a target problem is set, memory will be allocated accordingly in the constructor
    static void SetTargetProblem(const BProblem& p) { target_problem_ = &p; }
    static const BProblem& TargetProblem();


private:
    TDecVec variables_;
    TObjVec objectives_;
    TObjVec converted_objectives_;

    static const BProblem* target_problem_;
};



std::ostream& operator << (std::ostream& os, const CIndividual& indv);

#endif

#ifndef INITIALIZATION__
#define INITIALIZATION__


// ----------------------------------------------------------------------
//                        CRandomInitialization
// ----------------------------------------------------------------------

class CIndividual;
class CPopulation;
class BProblem;

class CRandomInitialization
{
public:
    void operator()(CPopulation* pop, const BProblem& prob) const;
    void operator()(CIndividual* indv, const BProblem& prob) const;
};

extern CRandomInitialization RandomInitialization;

#endif

#ifndef POPULATION__
#define POPULATION__

// #include "individual.h"

#include <vector>

class CPopulation
{
public:
    explicit CPopulation(std::size_t s = 0) :individuals_(s) {}
```

```
	CIndividual& operator[](std::size_t i) { return individuals_[i]; }
	const CIndividual& operator[](std::size_t i) const { return individuals_[i]; }

	std::size_t size() const { return individuals_.size(); }
	void resize(size_t t) { individuals_.resize(t); }
	void push_back(const CIndividual& indv) { individuals_.push_back(indv); }
	void clear() { individuals_.clear(); }

private:
	std::vector<CIndividual> individuals_;
};

#endif

#ifndef ENVIRONMENTAL_SELECTION__
#define ENVIRONMENTAL_SELECTION__
#include <vector>


// ----------------------------------------------------------------------
//		The environmental selection mechanism is the key innovation of
//  the GA algorithm.
//
//  Check Algorithm I in the original paper of NSGA-III.
// ----------------------------------------------------------------------

class CPopulation;
class CReferencePoint;

void SurvivorSelection(CPopulation* pnext, // population in the next generation
	CPopulation* pcur,  // population in the current generation
	size_t PopSize);

#endif

#ifndef LOG__
#define LOG__

#include <string>
#include <fstream>  // Include <fstream> for std::ios_base and std::ios_base::openmode

class CPopulation;
class Gnuplot;

// Save a population into the designated file.
bool SaveToFile(const std::string& fname, const CPopulation& pop, std::ios_base::openmode mode = std::ios_base::app);

// Show a population by calling gnuplot.
bool ShowPopulation(Gnuplot& gplot, const CPopulation&, const std::string& legend = "pop");

#endif

#ifndef MUTATION__
#define MUTATION__

// ---------------------------------------------------------------------------------
//			CPolynomialMutation : polynomial mutation
// ---------------------------------------------------------------------------------

class CIndividual;

class CPolynomialMutation
```

```cpp
{
public:
	explicit CPolynomialMutation(double mr = 0.0, double eta = 20) :mr_(mr), eta_(eta) {}

	void SetMutationRate(double mr) { mr_ = mr; }
	double MutationRate() const { return mr_; }
	void SetDistributionIndex(double eta) { eta_ = eta; }
	double DistributionIndex() const { return eta_; }

	bool operator()(CIndividual* c, double mr, double eta) const;
	bool operator()(CIndividual* c) const
	{
		return operator()(c, mr_, eta_);
	}

private:
	double mr_, // mutation rate
		eta_; // distribution index
};

#endif

#ifndef GA__
#define GA__

#include <cstddef>
#include <string>

// -------------------------------------------------------------------------------

class BProblem;
class CPopulation;

class CGA
{
public:
	explicit CGA(const std::string& inifile_name = "");
	void Solve(CPopulation* solutions, const BProblem& prob);

	const std::string& name() const { return name_; }
private:
	std::string name_;
	std::size_t obj_division_p_;
	std::size_t gen_num_;
	double  pc_, // crossover rate
		pm_, // mutation rate
		eta_c_, // eta in SBX
		eta_m_; // eta in Polynomial Mutation
};


#endif

// #include "ga.h"
// #include "base_problem.h"
// #include "individual.h"
// #include "population.h"

// #include "initialization.h"
// #include "crossover.h"
// #include "mutation.h"
// #include "survivor_selection.h"
```

```
// #include "gnuplot_interface.h"
// #include "log.h"
#include "windows.h" // for Sleep()

#include <vector>
#include <fstream>
#include<iostream>

using namespace std;

CGA::CGA(const string& inifile_name) :
	name_("GA"),
	obj_division_p_(12),
	gen_num_(1000),
	pc_(1.0), // default setting in GA
	eta_c_(30), // default setting
	eta_m_(20) // default setting
{
	if (inifile_name == "") return;

	ifstream inifile(inifile_name);
	if (!inifile) return;

	string dummy;
	inifile >> dummy >> dummy >> name_;
	inifile >> dummy >> dummy >> obj_division_p_;
	inifile >> dummy >> dummy >> gen_num_;
	inifile >> dummy >> dummy >> pc_;
	inifile >> dummy >> dummy >> eta_c_;
	inifile >> dummy >> dummy >> eta_m_;
}
// ----------------------------------------------------------------------
void CGA::Solve(CPopulation* solutions, const BProblem& problem)
{
	CIndividual::SetTargetProblem(problem);


	size_t PopSize = 50;
	while (PopSize % 4) PopSize += 1;

	//CPopulation pop[2] = { CPopulation(PopSize) };
	std::vector<CPopulation> pop(2, CPopulation(PopSize));
	CSimulatedBinaryCrossover SBX(pc_, eta_c_);
	CPolynomialMutation PolyMut(1.0 / problem.num_variables(), eta_m_);

	Gnuplot gplot;

	int cur = 0, next = 1;

	//std::cout << problem.lower_bounds()[0] << '\n';
	RandomInitialization(&pop[cur], problem);
	for (size_t i = 0; i < PopSize; i += 1)
	{
		problem.Evaluate(&pop[cur][i]);
	}

	for (size_t t = 0; t < gen_num_; t += 1)
	{
		pop[cur].resize(PopSize * 2);

		for (size_t i = 0; i < PopSize; i += 2)
		{
			int father = rand() % PopSize,
```

```
					mother = rand() % PopSize;

				SBX(&pop[cur][PopSize + i], &pop[cur][PopSize + i + 1], pop[cur][father], pop[cur][mother]);

				PolyMut(&pop[cur][PopSize + i]);
				PolyMut(&pop[cur][PopSize + i + 1]);

				problem.Evaluate(&pop[cur][PopSize + i]);
				problem.Evaluate(&pop[cur][PopSize + i + 1]);
			}

			SurvivorSelection(&pop[next], &pop[cur], PopSize);


			//ShowPopulation(gplot, pop[next], "pop"); Sleep(10);

			std::swap(cur, next);
		}

		*solutions = pop[cur];
}


// #include "comparator.h"
// #include "individual.h"

// ---------------------------------------------------------------------------------

CParetoDominate ParetoDominate;

// ---------------------------------------------------------------------------------
//							CParetoDominate
// ---------------------------------------------------------------------------------
bool CParetoDominate::operator()(const CIndividual& l, const CIndividual& r) const
{
		bool better = false;
		for (size_t f = 0; f < l.objs().size(); f += 1)
		{
			if (l.objs()[f] > r.objs()[f])
				return false;
			else if (l.objs()[f] < r.objs()[f])
				better = true;
		}

		return better;

}// CParetoDominate::operator()
// ---------------------------------------------------------------------------------

#include <vector>
// #include "math_aux.h"
#include<limits>

using namespace std;

namespace MathAux
{

// ---------------------------------------------------------------------
// ASF: Achivement Scalarization Function
// ---------------------------------------------------------------------
double ASF(const vector<double> &objs, const vector<double> &weight)
{
```

```
	double max_ratio = -numeric_limits<double>::max();
	for (size_t f=0; f<objs.size(); f+=1)
	{
		double w = weight[f]?weight[f]:0.00001;
		max_ratio = std::max(max_ratio, objs[f]/w);
	}
	return max_ratio;
}
// ---------------------------------------------------------------------



}// namespace MathAux



// #include "individual.h"
// #include "base_problem.h"

using std::size_t;


const BProblem* CIndividual::target_problem_ = 0;
// ----------------------------------------------------------------------
CIndividual::CIndividual(std::size_t num_vars, std::size_t num_objs) :
	variables_(num_vars),
	objectives_(num_objs),
	converted_objectives_(num_objs)
{
	if (target_problem_ != 0)
	{
		variables_.resize(target_problem_->num_variables());
		objectives_.resize(target_problem_->num_objectives());
		converted_objectives_.resize(target_problem_->num_objectives());
	}
}
// ----------------------------------------------------------------------
const BProblem& CIndividual::TargetProblem() { return *target_problem_; }
// ----------------------------------------------------------------------
std::ostream& operator << (std::ostream& os, const CIndividual& indv)
{
	for (size_t i = 0; i < indv.vars().size(); i += 1)
	{
		os << indv.vars()[i] << ' ';
	}

	os << " => ";
	for (size_t f = 0; f < indv.objs().size(); f += 1)
	{
		os << indv.objs()[f] << ' ';
	}

	return os;
}

// ----------------------------------------------------------------------

// #include "initialization.h"
// #include "base_problem.h"
// #include "individual.h"
// #include "population.h"
// #include "math_aux.h"
```

```
#include <cstddef>
#include<iostream>
using std::size_t;

CRandomInitialization RandomInitialization;

void CRandomInitialization::operator()(CIndividual* indv, const BProblem& prob) const
{
	CIndividual::TDecVec& x = indv->vars();
	x.resize(prob.num_variables());
	for (size_t i = 0; i < x.size(); i += 1)
	{
		x[i] = MathAux::random(prob.lower_bounds()[i], prob.upper_bounds()[i]);
	}

}
// ----------------------------------------------------------------------
void CRandomInitialization::operator()(CPopulation* pop, const BProblem& prob) const
{
	for (size_t i = 0; i < pop->size(); i += 1)
	{
		(*this)(&(*pop)[i], prob);
	}
}



// #include "crossover.h"
// #include "individual.h"
// #include "math_aux.h"
// #include "base_problem.h"

#include <cmath>
#include <algorithm>
#include <cstddef>
using std::size_t;

// ----------------------------------------------------------------------
// The implementation was adapted from the code of function realcross() in crossover.c
// http://www.iitk.ac.in/kangal/codes/nsga2/nsga2-gnuplot-v1.1.6.tar.gz
//
// ref: http://www.slideshare.net/paskorn/simulated-binary-crossover-presentation#
// ----------------------------------------------------------------------
double CSimulatedBinaryCrossover::get_betaq(double rand, double alpha, double eta) const
{
	double betaq = 0.0;
	if (rand <= (1.0 / alpha))
	{
		betaq = std::pow((rand * alpha), (1.0 / (eta + 1.0)));
	}
	else
	{
		betaq = std::pow((1.0 / (2.0 - rand * alpha)), (1.0 / (eta + 1.0)));
	}
	return betaq;
}
// ----------------------------------------------------------------------
bool CSimulatedBinaryCrossover::operator()(CIndividual* child1,
	CIndividual* child2,
	const CIndividual& parent1,
	const CIndividual& parent2,
	double cr,
```

```
	double eta) const
{
	*child1 = parent1;
	*child2 = parent2;

	if (MathAux::random(0.0, 1.0) > cr) return false; // not crossovered

	CIndividual::TDecVec& c1 = child1->vars(), & c2 = child2->vars();
	const CIndividual::TDecVec& p1 = parent1.vars(), & p2 = parent2.vars();

	for (size_t i = 0; i < c1.size(); i += 1)
	{
		if (MathAux::random(0.0, 1.0) > 0.5) continue; // these two variables are not crossovered
		if (std::fabs(p1[i] - p2[i]) <= MathAux::EPS) continue; // two values are the same

		double y1 = std::min(p1[i], p2[i]),
			y2 = std::max(p1[i], p2[i]);

		double lb = CIndividual::TargetProblem().lower_bounds()[i],
			ub = CIndividual::TargetProblem().upper_bounds()[i];

		double rand = MathAux::random(0.0, 1.0);

		// child 1
		double beta = 1.0 + (2.0 * (y1 - lb) / (y2 - y1)),
			alpha = 2.0 - std::pow(beta, -(eta + 1.0));
		double betaq = get_betaq(rand, alpha, eta);

		c1[i] = 0.5 * ((y1 + y2) - betaq * (y2 - y1));

		// child 2
		beta = 1.0 + (2.0 * (ub - y2) / (y2 - y1));
		alpha = 2.0 - std::pow(beta, -(eta + 1.0));
		betaq = get_betaq(rand, alpha, eta);

		c2[i] = 0.5 * ((y1 + y2) + betaq * (y2 - y1));

		// boundary checking
		c1[i] = std::min(ub, std::max(lb, c1[i]));
		c2[i] = std::min(ub, std::max(lb, c2[i]));

		if (MathAux::random(0.0, 1.0) <= 0.5)
		{
			std::swap(c1[i], c2[i]);
		}
	}

	return true;
}// CSimulatedBinaryCrossover



// #include "mutation.h"
// #include "individual.h"
// #include "math_aux.h"
// #include "base_problem.h"

#include <cstddef>
#include <algorithm>
using std::size_t;

// ----------------------------------------------------------------------
// The implementation was adapted from the code of function real_mutate_ind() in mutation.c in
```

```
// http://www.iitk.ac.in/kangal/codes/nsga2/nsga2-gnuplot-v1.1.6.tar.gz
//
// ref: http://www.slideshare.net/paskorn/simulated-binary-crossover-presentation#
// ------------------------------------------------------------------
bool CPolynomialMutation::operator()(CIndividual* indv, double mr, double eta) const
{
    //int j;
    //double rnd, delta1, delta2, mut_pow, deltaq;
    //double y, yl, yu, val, xy;

    bool mutated = false;

    CIndividual::TDecVec& x = indv->vars();

    for (size_t i = 0; i < x.size(); i += 1)
    {
        if (MathAux::random(0.0, 1.0) <= mr)
        {
            mutated = true;

            double y = x[i],
                lb = CIndividual::TargetProblem().lower_bounds()[i],
                ub = CIndividual::TargetProblem().upper_bounds()[i];

            double delta1 = (y - lb) / (ub - lb),
                delta2 = (ub - y) / (ub - lb);

            double mut_pow = 1.0 / (eta + 1.0);

            double rnd = MathAux::random(0.0, 1.0), deltaq = 0.0;
            if (rnd <= 0.5)
            {
                double xy = 1.0 - delta1;
                double val = 2.0 * rnd + (1.0 - 2.0 * rnd) * (pow(xy, (eta + 1.0)));
                deltaq = pow(val, mut_pow) - 1.0;
            }
            else
            {
                double xy = 1.0 - delta2;
                double val = 2.0 * (1.0 - rnd) + 2.0 * (rnd - 0.5) * (pow(xy, (eta + 1.0)));
                deltaq = 1.0 - (pow(val, mut_pow));
            }

            y = y + deltaq * (ub - lb);
            y = std::min(ub, std::max(lb, y));

            x[i] = y;
        }
    }

    return mutated;
}// CPolynomialMutation

// #include "survivor_selection.h"
// #include "population.h"
// #include "math_aux.h"

#include <limits>
#include <algorithm>

using namespace std;

// ------------------------------------------------------------------
```

```
// SurvivorSelection():
// ---------------------------------------------------------------------
void SurvivorSelection(CPopulation *pnext, CPopulation *pcur,  size_t PopSize)
{
	CPopulation &cur = *pcur, &next = *pnext;
	next.clear();
	std::vector<size_t> index;
	for (int i = 0; i < cur.size(); i++)index.push_back(i);


	std::sort(index.begin(), index.end(), [&](size_t a, size_t b) {
		return cur[a].objs()[0]+ cur[a].objs()[1] < cur[b].objs()[0]+ cur[b].objs()[1];
		});


	for (size_t t = 0; t < PopSize ; t += 1)
	{
		 next.push_back(cur[index[t]]);
	}

}
// ---------------------------------------------------------------------



// #include "gnuplot_interface.h"
#include <iostream>
#include <sstream>

using namespace std;

// --------------------------------------------------------
// Ref:
// http://user.frdm.info/ckhung/b/ma/gnuplot.php
// --------------------------------------------------------

Gnuplot::Gnuplot()
{
	// with -persist option you will see the windows as your program ends
	//gnuplotpipe=_popen("gnuplot -persist","w");
	//without that option you will not see the window

	// because I choose the terminal to output files so I don't want to see the window

	gnuplotpipe = _popen("gnuplot", "w");

	if (!gnuplotpipe)
	{
		cerr << ("Gnuplot not found !");
	}
}
// --------------------------------------------------------
Gnuplot::~Gnuplot()
{
	fprintf(gnuplotpipe, "exit\n");
	_pclose(gnuplotpipe);
}
// --------------------------------------------------------
void Gnuplot::operator()(const string& command)
{
	fprintf(gnuplotpipe, "%s\n", command.c_str());
	fflush(gnuplotpipe);
	// flush is necessary, nothing gets plotted else
```

```cpp
}
// --------------------------------------------------------
void Gnuplot::set_title(const std::string& title)
{
	ostringstream ss;
	ss << "set title \"" << title << "\"";
	operator()(ss.str());
}
// --------------------------------------------------------
void Gnuplot::plot(const std::string& fname, std::size_t x, std::size_t y)
{
	ostringstream ss;

	ss << "plot \"" << fname << "\" using " << x << ":" << y;
	operator()(ss.str());
}
// --------------------------------------------------------
void Gnuplot::splot(const std::string& fname, std::size_t x, std::size_t y, std::size_t z)
{
	ostringstream ss;

	ss << "splot \"" << fname << "\" using " << x << ":" << y << ":" << z;
	operator()(ss.str());
}
// --------------------------------------------------------


// #include "assignment_problem.h"
// #include "individual.h"
// #include "math_aux.h"

#include <cmath>
#include <vector>
#include<iostream>

using std::size_t;
using std::cos;

// ------------------------------------------------------------------
//                    CProblemAssignment
// ------------------------------------------------------------------

CProblemAssignment::CProblemAssignment(size_t M, size_t k, const std::string& name, double lbs, double ubs) :
	BProblem(name),
	M_(M),
	k_(k)
{
	lbs_.resize( k_, lbs); // lower bound
	ubs_.resize( k_, ubs); // upper bound
}


bool CProblemAssignment5::Evaluate(CIndividual* indv) const
{
	CIndividual::TDecVec& x = indv->vars();
	CIndividual::TObjVec& f = indv->objs();

	if (x.size() != k_) return false; // #variables does not match

	f.resize(M_, 0);
	for (size_t i = 1; i < k_; ++i) {
	f[0] += -10 * exp(-0.2 * sqrt(MathAux::square(x[i - 1]) + MathAux::square(x[i])));
	}
```

```
	for (size_t i = 0; i < k_; ++i) {
	f[1] += pow(abs(x[i]), 0.8) + 5 * sin(pow(x[i], 3));
	}


	return true;
}


bool CProblemAssignment6::Evaluate(CIndividual* indv) const
{
	CIndividual::TDecVec& x = indv->vars();
	CIndividual::TObjVec& f = indv->objs();
	//std::cout << (x.size() == k_)<<'\n';
	if (x.size() != k_) return false; // #variables does not match

	f.resize(M_, 0);
	f[0] = MathAux::square(x[0]);
	f[1] = MathAux::square(x[0] - 2);

	return true;
}



// #include "log.h"
// #include "population.h"
// #include "gnuplot_interface.h"

#include <fstream>
#include<iostream>
using namespace std;

#define OUTPUT_DECISION_VECTOR

bool SaveToFile(const std::string& fname, const CPopulation& pop, ios_base::openmode mode)
{
	ofstream ofile(fname.c_str(), mode);
	if (!ofile) return false;

	for (size_t i = 0; i < pop.size(); i += 1)
	{
#ifdef OUTPUT_DECISION_VECTOR

		for (size_t j = 0; j < pop[i].vars().size(); j += 1)
		{
			ofile << pop[i].vars()[j] << ' ';
		}
#endif
		//std::cout << pop[i].objs().size() << endl;
		for (size_t f = 0; f < pop[i].objs().size(); f += 1)
		{
			ofile << pop[i].objs()[f] << ' ';
		}

	    ofile << pop[i].objs()[0] + pop[i].objs()[1] << ' ';

	    ofile << endl;
	}
	ofile << endl;
	return true;
}
// --------------------------------------------------------------------
```

```cpp
bool ShowPopulation(Gnuplot& gplot, const CPopulation& pop, const std::string& legend)
{
	if (!SaveToFile(legend, pop, ios_base::out)) return false;


	size_t n = 0;
#ifdef OUTPUT_DECISION_VECTOR
	n = pop[0].vars().size();
#endif

	if (pop[0].objs().size() == 2)
	{
		gplot.plot(legend, n + 1, n + 2);
		return true;
	}
	else if (pop[0].objs().size() == 3)
	{
		gplot.splot(legend, n + 1, n + 2, n + 3);
		return true;
	}
	else
		return false;
}


// #include "assignment_problem.h"
// #include "ga.h"
// #include "population.h"
// #include "gnuplot_interface.h"
// #include "log.h"

#include <ctime>
#include <cstdlib>
#include <iostream>

// #include "individual.h"
// #include "math_aux.h"

using namespace std;

int main()
{
	CGA ga("ga");
	CPopulation solutions;
	Gnuplot gplot;

	const size_t NumRuns = 10;

	BProblem* problem5 = new CProblemAssignment5(2, 3,-5,5);
	BProblem* problem6 = new CProblemAssignment6(2, 1,-10,10);

	BProblem* problem = problem5;

	//std::cout << problem->upper_bounds()[0] << '\n';

	for (size_t r = 0; r < NumRuns; r += 1)
	{
		srand(r); cout << "Run Number: " << r << endl;

		ga.Solve(&solutions, *problem);
		SaveToFile(ga.name() + "-" + problem->name() + ".txt", solutions);
		ShowPopulation(gplot, solutions, "pop"); system("pause");
	}
```

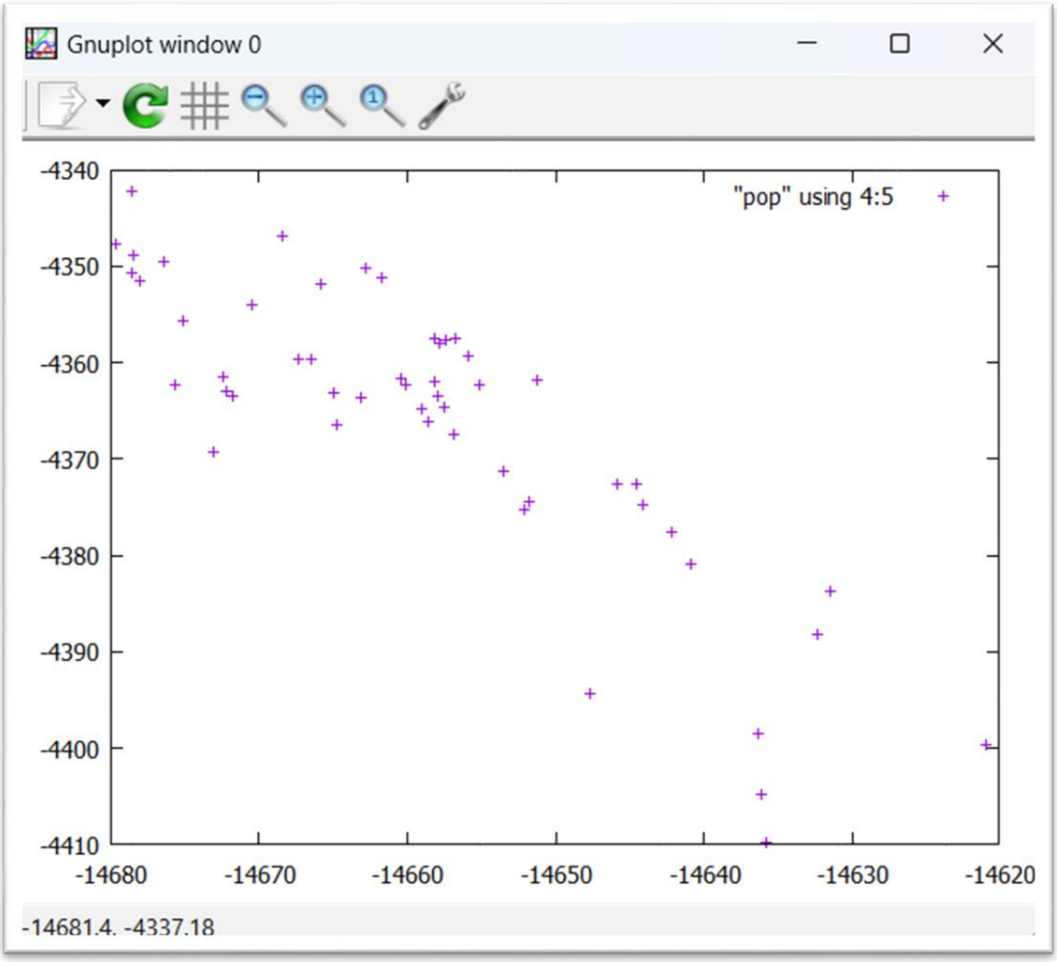
```
        delete problem;

        return 0;
}
```

Input: Terminal Input is not required Everything is Automated. If you want to change the parameters please update the main function and ga function.

You can also add ga.ini file and write parameter values.

Output:

Values of F1 and F2 after max iteration



The Values after the max iteration (x1, x2, x3, f1, f2, f1+f2)

```
1432   -0.848038 0.306477 -1.02812 -14582.7 -4192.69 -18775.4
1433   -1.22531 0.184394 -0.949017 -14602.5 -4172.09 -18774.6
1434   -1.19753 -0.23692 -1.14856 -14599.6 -4174.73 -18774.3
1435   -1.19753 0.436967 -1.25045 -14599.2 -4173.62 -18772.8
1436   -1.21293 -0.306357 0.277976 -14582 -4189.78 -18771.8
1437   -0.897956 -0.0889139 0.0948888 -14602.7 -4167.45 -18770.1
1438   0.176083 -1.20774 -0.966965 -14569.9 -4199.96 -18769.9
1439   -1.19753 0.644352 -0.213169 -14600.2 -4168.84 -18769
1440   -1.42717 -0.304403 0.134731 -14581.9 -4186.15 -18768
1441   0.859278 -0.701795 -0.972425 -14600.5 -4165.44 -18765.9
1442   -0.566315 -0.940671 -0.359599 -14564.6 -4200.24 -18764.8
1443   0.1235 -0.954392 -0.328125 -14564.8 -4199.86 -18764.7
1444   -0.835569 -0.982542 -0.785208 -14582.2 -4182.4 -18764.6
1445   -0.958815 -0.979098 -0.568919 -14582.2 -4182.15 -18764.4
1446   0.403137 -0.680126 -0.97398 -14565.9 -4198.25 -18764.2
1447   0.404212 -0.759389 -1.96727 -14564.5 -4198.88 -18763.3
1448   -1.16672 -0.957134 -0.748334 -14573.7 -4189.63 -18763.3
1449   -0.284092 -1.15004 -1.2069 -14564.8 -4197.01 -18761.9
1450   -2.06595 -0.17444 -0.746769 -14567.2 -4194.58 -18761.8
1451   -2.07127 -0.810606 -0.96771 -14572.6 -4187.07 -18759.7
1452   -0.836532 -1.24748 -1.21439 -14562.8 -4196.57 -18759.3
1453   -0.716354 -1.13208 -1.206 -14573.9 -4184.79 -18758.7
1454   -1.16671 -0.845286 -0.357636 -14564.3 -4193.6 -18757.9
1455   0.757424 -0.288846 -0.57655 -14576.1 -4181.83 -18757.9
1456   -0.865662 -0.304403 -0.862775 -14564.9 -4191.49 -18756.4
```

Hands On calculations:

Number of populations =4
Tournament parent selection
SBX crossover
Polynomial Mutation
Survivor selection Fitness propionate

Generation Number =1

Parent selection

Sr. No.	x1	x2	x3	f1(x)	f2(x)	g(x)
1	-4.9884	-2.64428	1.48152	-8.68714	13.3724	4.68527
2	-4.25626	-2.29759	-1.39973	-9.63955	1.66223	-7.97732
3	-2.44713	4.8529	-1.81082	-6.92117	9.18446	2.26329
4	4.34233	-3.21375	3.5757	-7.21749	9.56758	2.35009

Crossover

Father	Mother			c1				c2	
			x1	x2	x3		x1	x2	x3
3	2	1	-4.23878	-2.28495	-1.81082	2	-2.46461	4.82254	-1.39973
3	3	3	-2.44713	4.8529	-1.81082	4	-2.44713	4.8529	-1.81082

Mutation

		Mutated Value				
Sr.	x1	x2	x3	f1(x)	f2(x)	g(x)

No.						
1	-4.23878	-2.28495	-1.81082	-16.32 17	17.1229	0.802917
2	-1.80237	4.57723	-1.39973	-17.2175	13.0664	-4.15109
3	-2.44713	4.8529	-1.81082	-13.8423	18.3689	4.52657
4	-1.70358	4.95464	-1.81082	-13.9098	26.3758	12.466

Survivor Selection

Sr. No.	x1	x2	x3	g(x)		Next Generation
		Old Population				
1	-4.9884	-2.64428	1.48152	4.68527	1	2
2	-4.25626	-2.29759	-1.39973	-7.97732	2	6
3	-2.44713	4.8529	-1.81082	2.26329	3	5
4	4.34233	-3.21375	3.5757	2.35009	4	3
		New Population				
5	-4.23878	-2.28495	-1.81082	0.802917		
6	-1.80237	4.57723	-1.39973	-4.15109		
7	-2.44713	4.8529	-1.81082	4.52657		
8	-1.70358	4.95464	-1.81082	12.466		

Generation Number =2

Parent selection

Sr. No.	x1	x2	x3	f1(x)	f2(x)	g(x)
1	-4.25626	-2.29759	-1.39973	-9.63955	1.66223	-7.97732
2	-1.80237	4.57723	-1.39973	-17.2175	13.0664	-4.15109
3	-4.23878	-2.28495	-1.81082	-16.32 17	17.1229	0.802917
4	-2.44713	4.8529	-1.81082	-6.92117	9.18446	2.26329

Crossover

Father	Mother			c1				c2	
			x1	x2	x3		x1	x2	x3
2	1	1	-1.80237	4.5803	-1.39973	2	-4.25626	-2.30231	-1.39973
1	3	3	-4.25626	-2.28487	-1.39973	4	-4.23878	-2.29767	-1.81082

Mutation

		Mutated Value				
Sr.	x1	x2	x3	f1(x)	f2(x)	g(x)

No.						
1	-1.80237	4.5803	-1.39973	-24.7911	24.3035	-0.487555
2	-4.25626	-2.30231	-2.28626	-18.6648	8.44429	-10.2205
3	-4.25626	-1.94899	-1.39973	-25.7147	24.8487	-23.2843
4	-4.23878	-2.28829	-1.81082	-25.7147	24.8487	-0.866012

Survivor Selection

Sr. No.	x1	x2	x3	g(x)		Next Generation
		Old Population				
1	-4.25626	-2.29759	-1.39973	-7.97732	1	7
2	-1.80237	4.57723	-1.39973	-4.15109	2	6
3	-4.23878	-2.28495	-1.81082	0.802917	3	1
4	-2.44713	4.8529	-1.81082	2.26329	4	2
		New Population				
5	-1.80237	4.5803	-1.39973	-0.487555		
6	-4.25626	-2.30231	-2.28626	-10.2205		
7	-4.25626	-1.94899	-1.39973	-23.2843		
8	-4.23878	-2.28829	-1.81082	-0.866012		

Best Answer: x = [-4.25626 -1.94899 -1.39973]

C++ code for the NSGAII:

```
#ifndef MATH_AUX__
#define MATH_AUX__

#include <cstdlib>
#include <vector>

namespace MathAux
{
	const double PI = 3.1415926;
	const double EPS = 1.0e-14; // follow nsga-ii source code
	inline double square(double n) { return n * n; }
	inline double random(double lb, double ub) { return lb + (static_cast<double>(std::rand()) / RAND_MAX) * (ub - lb); }

	// ASF(): achievement scalarization function
	double ASF(const std::vector<double>& objs, const std::vector<double>& weight);
}

#endif


#ifndef BASE_PROBLEM__
#define BASE_PROBLEM__

#include <string>
#include <vector>
```

```cpp
// ----------------------------------------------------------------------
//                    BProblem: the base class of problems (e.g. ZDT and DTLZ)
// ----------------------------------------------------------------------
class CIndividual;

class BProblem
{
public:
        explicit BProblem(const std::string& name) :name_(name) {}
        virtual ~BProblem() {}

        virtual std::size_t num_variables() const = 0;
        virtual std::size_t num_objectives() const = 0;
        virtual bool Evaluate(CIndividual* indv) const = 0;

        const std::string& name() const { return name_; }
        const std::vector<double>& lower_bounds() const { return lbs_; }
        const std::vector<double>& upper_bounds() const { return ubs_; }

protected:
        std::string name_;

        std::vector<double> lbs_, // lower bounds of variables
                ubs_; // upper bounds of variables
};

#endif


#ifndef ASSIGNMENT_PROBLEM__
#define ASSIGNMENT_PROBLEM__

//#include "base_problem.h"
#include <cstddef>
#include<iostream>

// ----------------------------------------------------------------------
class CProblemAssignment : public BProblem
{
public:
        CProblemAssignment(std::size_t M, std::size_t k, const std::string& name, double lbs, double ubs);

        virtual std::size_t num_variables() const { return k_; }
        virtual std::size_t num_objectives() const { return M_; }

        virtual bool Evaluate(CIndividual* indv) const = 0;

protected:
        std::size_t M_; // number of objectives
        std::size_t k_; // number of variables
};


// ----------------------------------------------------------------------
//                    CProblemAssignment5
// ----------------------------------------------------------------------

class CProblemAssignment5 : public CProblemAssignment
{
public:
        explicit CProblemAssignment5(std::size_t M, std::size_t k, double lbs, double ubs) :CProblemAssignment(M, k, "Assignment5",
lbs, ubs) {}
```

```
	virtual bool Evaluate(CIndividual* indv) const;
};


// ---------------------------------------------------------------------
//			CProblemAssignment6
// ---------------------------------------------------------------------

class CProblemAssignment6 : public CProblemAssignment
{
public:
	explicit CProblemAssignment6(std::size_t M, std::size_t k, double lbs, double ubs) :CProblemAssignment(M, k, "Assignment6", lbs, ubs) {
		//std::cout << "what happend !" << lbs << '\n';
	}
	virtual bool Evaluate(CIndividual* indv) const;
};

#endif

#ifndef COMPARATOR__
#define COMPARATOR__

class CIndividual;

// ------------------------------------------------------------------------------
//				BComparator : the base class of comparison operators
// ------------------------------------------------------------------------------
class BComparator
{
public:
	virtual ~BComparator() {}

	virtual bool	operator()(const CIndividual& l, const CIndividual& r) const = 0;
};



// ------------------------------------------------------------------------------
//				CParetoDominate
// ------------------------------------------------------------------------------

class CParetoDominate : public BComparator
{
public:
	virtual	bool	operator()(const CIndividual& l, const CIndividual& r) const;
};


extern CParetoDominate ParetoDominate;
#endif


#ifndef CROSSOVER__
#define CROSSOVER__

// ------------------------------------------------------------------------------
//			CSimulatedBinaryCrossover : simulated binary crossover (SBX)
// ------------------------------------------------------------------------------


class CIndividual;
class CSimulatedBinaryCrossover
```

```
{
public:
	explicit CSimulatedBinaryCrossover(double cr = 1.0, double eta = 30) :cr_(cr), eta_(eta) {} // NSGA-III (t-EC 2014) setting

	void SetCrossoverRate(double cr) { cr_ = cr; }
	double CrossoverRate() const { return cr_; }
	void SetDistributionIndex(double eta) { eta_ = eta; }
	double DistributionIndex() const { return eta_; }

	bool operator()(CIndividual* c1, CIndividual* c2, const CIndividual& p1, const CIndividual& p2, double cr, double eta) const;
	bool operator()(CIndividual* c1, CIndividual* c2, const CIndividual& p1, const CIndividual& p2) const
	{
		return operator()(c1, c2, p1, p2, cr_, eta_);
	}

private:

	double get_betaq(double rand, double alpha, double eta) const;

	double cr_; // crossover rate
	double eta_; // distribution index
};


#endif


#include <string>
#include <stdio.h>

#include <string>
#include <cstddef>

// ----------------------------------------------------------------------
// Gnuplot
//
// This is just a very simple interface to call gnuplot in the program.
// Now it seems to work only under windows + visual studio.
// ----------------------------------------------------------------------

class Gnuplot
{
public:
	Gnuplot();
	~Gnuplot();

	// prohibit copying (VS2012 does not support 'delete')
	Gnuplot(const Gnuplot&);
	Gnuplot& operator=(const Gnuplot&);

	// send any command to gnuplot
	void operator ()(const std::string& command);

	void reset() { operator()("reset"); }
	void replot() { operator()("replot"); }
	void set_title(const std::string& title);

	void plot(const std::string& fname, std::size_t x, std::size_t y);
	void splot(const std::string& fname, std::size_t x, std::size_t y, std::size_t z);

protected:
	FILE* gnuplotpipe;
};
```

```cpp
#ifndef INDIVIDUAL__
#define INDIVIDUAL__

#include <vector>
#include <ostream>

// --------------------------------------------------------------------
//                      CIndividual
// --------------------------------------------------------------------

class BProblem;

class CIndividual
{
public:
	typedef double TGene;
	typedef std::vector<TGene> TDecVec;
	typedef std::vector<double> TObjVec;

	explicit CIndividual(std::size_t num_vars = 0, std::size_t num_objs = 0);

	TDecVec& vars() { return variables_; }
	const TDecVec& vars() const { return variables_; }

	TObjVec& objs() { return objectives_; }
	const TObjVec& objs() const { return objectives_; }

	TObjVec& conv_objs() { return converted_objectives_; }
	const TObjVec& conv_objs() const { return converted_objectives_; }

	// if a target problem is set, memory will be allocated accordingly in the constructor
	static void SetTargetProblem(const BProblem& p) { target_problem_ = &p; }
	static const BProblem& TargetProblem();


private:
	TDecVec variables_;
	TObjVec objectives_;
	TObjVec converted_objectives_;

	static const BProblem* target_problem_;
};



std::ostream& operator << (std::ostream& os, const CIndividual& indv);

#endif

#ifndef INITIALIZATION__
#define INITIALIZATION__


// --------------------------------------------------------------------
//                      CRandomInitialization
// --------------------------------------------------------------------

class CIndividual;
class CPopulation;
class BProblem;

class CRandomInitialization
```

```
{
public:
	void operator()(CPopulation* pop, const BProblem& prob) const;
	void operator()(CIndividual* indv, const BProblem& prob) const;
};

extern CRandomInitialization RandomInitialization;

#endif


#ifndef POPULATION__
#define POPULATION__

// #include "individual.h"

#include <vector>

class CPopulation
{
public:
	explicit CPopulation(std::size_t s = 0) :individuals_(s) {}

	CIndividual& operator[](std::size_t i) { return individuals_[i]; }
	const CIndividual& operator[](std::size_t i) const { return individuals_[i]; }

	std::size_t size() const { return individuals_.size(); }
	void resize(size_t t) { individuals_.resize(t); }
	void push_back(const CIndividual& indv) { individuals_.push_back(indv); }
	void clear() { individuals_.clear(); }

private:
	std::vector<CIndividual> individuals_;
};

#endif

#ifndef ENVIRONMENTAL_SELECTION__
#define ENVIRONMENTAL_SELECTION__
#include <vector>


// ----------------------------------------------------------------------
//	The environmental selection mechanism is the key innovation of
//  the NSGA-III algorithm.
//
//  Check Algorithm I in the original paper of NSGA-III.
// ----------------------------------------------------------------------

class CPopulation;
class CReferencePoint;

void SurvivorSelection(CPopulation *pnext, // population in the next generation
										CPopulation *pcur,  // population in the current generation
										size_t PopSize);

#endif

#ifndef NONDOMINATED_SORT__
#define NONDOMINATED_SORT__

#include <vector>
```

```cpp
class BComparator;
class CPopulation;

class CNondominatedSort
{
public:
	explicit CNondominatedSort(const BComparator &d):dominate(d) {}

	// prohibit copying (VS2012 does not support 'delete')
	CNondominatedSort(const CNondominatedSort &);
	CNondominatedSort & operator= (const CNondominatedSort &);

	typedef std::vector<std::size_t> TFrontMembers; // a set of indices of individuals in a certain front
	typedef std::vector<TFrontMembers> TFronts; // a set of fronts

	std::pair< std::vector<size_t>,TFronts> operator()(const CPopulation &pop) const;

private:
	const BComparator &dominate;
};

extern CNondominatedSort NondominatedSort;

#endif

#ifndef LOG__
#define LOG__

#include <string>
#include <fstream>  // Include <fstream> for std::ios_base and std::ios_base::openmode

class CPopulation;
class Gnuplot;

// Save a population into the designated file.
bool SaveToFile(const std::string& fname, const CPopulation& pop, std::ios_base::openmode mode = std::ios_base::app);

// Show a population by calling gnuplot.
bool ShowPopulation(Gnuplot& gplot, const CPopulation&, const std::string& legend = "pop");

#endif

#ifndef MUTATION__
#define MUTATION__

// ---------------------------------------------------------------------------------
//			CPolynomialMutation : polynomial mutation
// ---------------------------------------------------------------------------------

class CIndividual;

class CPolynomialMutation
{
public:
	explicit CPolynomialMutation(double mr = 0.0, double eta = 20) :mr_(mr), eta_(eta) {}

	void SetMutationRate(double mr) { mr_ = mr; }
	double MutationRate() const { return mr_; }
	void SetDistributionIndex(double eta) { eta_ = eta; }
	double DistributionIndex() const { return eta_; }

	bool operator()(CIndividual* c, double mr, double eta) const;
	bool operator()(CIndividual* c) const
```

```
		{
			return operator()(c, mr_, eta_);
		}

private:
	double mr_, // mutation rate
		eta_; // distribution index
};

#endif

#ifndef CROWDING_DISTANCE_H
#define CROWDING_DISTANCE_H

// #include "nondominated_sort.h"
// #include "population.h"
#include <vector>

std:: vector<double> CalculateCrowdingDistance(CPopulation& cur, CNondominatedSort::TFronts& fronts);
void SortBasedOnCrowdingDistance(CPopulation& cur, CNondominatedSort::TFronts& fronts);

#endif

#ifndef NSGAII__
#define NSGAII__

#include <cstddef>
#include <string>

// ------------------------------------------------------------------------------
//	NSGAII
// Taken from NSGA III
// Deb and Jain, "An Evolutionary Many-Objective Optimization Algorithm Using
// Reference-point Based Non-dominated Sorting Approach, Part I: Solving Problems with
// Box Constraints," IEEE Transactions on Evolutionary Computation, to appear.
//
// http://dx.doi.org/10.1109/TEVC.2013.2281535
// ------------------------------------------------------------------------------

class BProblem;
class CPopulation;

class CNSGAII
{
public:
	explicit CNSGAII(const std::string& inifile_name = "");
	void Solve(CPopulation* solutions, const BProblem& prob);

	const std::string& name() const { return name_; }
private:
	std::string name_;
	std::size_t obj_division_p_;
	std::size_t gen_num_;
	double	pc_, // crossover rate
		pm_, // mutation rate
		eta_c_, // eta in SBX
		eta_m_; // eta in Polynomial Mutation
};


#endif

// end of headers
```

```cpp
// #include "nsgaii.h"
// #include "base_problem.h"
// #include "individual.h"
// #include "population.h"

// #include "initialization.h"
// #include "crossover.h"
// #include "mutation.h"
// #include "survivor_selection.h"

// #include "gnuplot_interface.h"
// #include "log.h"
#include "windows.h" // for Sleep()

#include <vector>
#include <fstream>
#include<iostream>

using namespace std;

CNSGAII::CNSGAII(const string& inifile_name) :
	name_("NSGAII"),
	obj_division_p_(12),
	gen_num_(1000),
	pc_(1.0), // default setting in NSGA-II
	eta_c_(30), // default setting
	eta_m_(20) // default setting
{
	if (inifile_name == "") return;

	ifstream inifile(inifile_name);
	if (!inifile) return;

	string dummy;
	inifile >> dummy >> dummy >> name_;
	inifile >> dummy >> dummy >> obj_division_p_;
	inifile >> dummy >> dummy >> gen_num_;
	inifile >> dummy >> dummy >> pc_;
	inifile >> dummy >> dummy >> eta_c_;
	inifile >> dummy >> dummy >> eta_m_;
}
// ----------------------------------------------------------------------
void CNSGAII::Solve(CPopulation* solutions, const BProblem& problem)
{
	CIndividual::SetTargetProblem(problem);


	size_t PopSize = 50;
	while (PopSize % 4) PopSize += 1;

	//CPopulation pop[2] = { CPopulation(PopSize) };
	std::vector<CPopulation> pop(2, CPopulation(PopSize));
	CSimulatedBinaryCrossover SBX(pc_, eta_c_);
	CPolynomialMutation PolyMut(1.0 / problem.num_variables(), eta_m_);

	Gnuplot gplot;

	int cur = 0, next = 1;

	//std::cout << problem.lower_bounds()[0] << '\n';
	RandomInitialization(&pop[cur], problem);
	for (size_t i = 0; i < PopSize; i += 1)
```

```
	{
		problem.Evaluate(&pop[cur][i]);
	}

	for (size_t t = 0; t < gen_num_; t += 1)
	{
		pop[cur].resize(PopSize * 2);

		for (size_t i = 0; i < PopSize; i += 2)
		{
			int father = rand() % PopSize,
				mother = rand() % PopSize;

			SBX(&pop[cur][PopSize + i], &pop[cur][PopSize + i + 1], pop[cur][father], pop[cur][mother]);

			PolyMut(&pop[cur][PopSize + i]);
			PolyMut(&pop[cur][PopSize + i + 1]);

			problem.Evaluate(&pop[cur][PopSize + i]);
			problem.Evaluate(&pop[cur][PopSize + i + 1]);

		}

		SurvivorSelection(&pop[next], &pop[cur], PopSize);

		//ShowPopulation(gplot, pop[next], "pop"); Sleep(10);

		std::swap(cur, next);
	}

	*solutions = pop[cur];
}

// #include "comparator.h"
// #include "individual.h"

// ----------------------------------------------------------------------------------

CParetoDominate ParetoDominate;

// ----------------------------------------------------------------------------------
//							CParetoDominate
// ----------------------------------------------------------------------------------
bool CParetoDominate::operator()(const CIndividual& l, const CIndividual& r) const
{
	bool better = false;
	for (size_t f = 0; f < l.objs().size(); f += 1)
	{
		if (l.objs()[f] > r.objs()[f])
			return false;
		else if (l.objs()[f] < r.objs()[f])
			better = true;
	}

	return better;

}// CParetoDominate::operator()
// ----------------------------------------------------------------------------------




#include <vector>
```

```cpp
#include <limits>
// #include "math_aux.h"

using namespace std;

namespace MathAux
{

// --------------------------------------------------------------------
// ASF: Achivement Scalarization Function
// --------------------------------------------------------------------
double ASF(const vector<double> &objs, const vector<double> &weight)
{
	double max_ratio = -numeric_limits<double>::max();
	for (size_t f=0; f<objs.size(); f+=1)
	{
		double w = weight[f]?weight[f]:0.00001;
		max_ratio = std::max(max_ratio, objs[f]/w);
	}
	return max_ratio;
}
// -------------------------------------------------------------------




}// namespace MathAux



// #include "individual.h"
// #include "base_problem.h"

using std::size_t;


const BProblem* CIndividual::target_problem_ = 0;
// --------------------------------------------------------------------
CIndividual::CIndividual(std::size_t num_vars, std::size_t num_objs) :
	variables_(num_vars),
	objectives_(num_objs),
	converted_objectives_(num_objs)
{
	if (target_problem_ != 0)
	{
		variables_.resize(target_problem_->num_variables());
		objectives_.resize(target_problem_->num_objectives());
		converted_objectives_.resize(target_problem_->num_objectives());
	}
}
// --------------------------------------------------------------------
const BProblem& CIndividual::TargetProblem() { return *target_problem_; }
// --------------------------------------------------------------------
std::ostream& operator << (std::ostream& os, const CIndividual& indv)
{
	for (size_t i = 0; i < indv.vars().size(); i += 1)
	{
		os << indv.vars()[i] << ' ';
	}

	os << " => ";
	for (size_t f = 0; f < indv.objs().size(); f += 1)
	{
```

```
			os << indv.objs()[f] << ' ';
		}

		return os;
}

// ----------------------------------------------------------------------

// #include "initialization.h"
// #include "base_problem.h"
// #include "individual.h"
// #include "population.h"
// #include "math_aux.h"

#include <cstddef>
using std::size_t;

CRandomInitialization RandomInitialization;

void CRandomInitialization::operator()(CIndividual* indv, const BProblem& prob) const
{
		CIndividual::TDecVec& x = indv->vars();
		x.resize(prob.num_variables());
		for (size_t i = 0; i < x.size(); i += 1)
		{
			x[i] = MathAux::random(prob.lower_bounds()[i], prob.upper_bounds()[i]);
		}
}
// ----------------------------------------------------------------------
void CRandomInitialization::operator()(CPopulation* pop, const BProblem& prob) const
{
		for (size_t i = 0; i < pop->size(); i += 1)
		{
			(*this)(&(*pop)[i], prob);
		}
}







// #include "crossover.h"
// #include "individual.h"
// #include "math_aux.h"
// #include "base_problem.h"

#include <cmath>
#include <algorithm>
#include <cstddef>
using std::size_t;

// ----------------------------------------------------------------------
// The implementation was adapted from the code of function realcross() in crossover.c
// http://www.iitk.ac.in/kangal/codes/nsga2/nsga2-gnuplot-v1.1.6.tar.gz
//
// ref: http://www.slideshare.net/paskorn/simulated-binary-crossover-presentation#
// ----------------------------------------------------------------------
double CSimulatedBinaryCrossover::get_betaq(double rand, double alpha, double eta) const
{
		double betaq = 0.0;
		if (rand <= (1.0 / alpha))
```

```
	{
		betaq = std::pow((rand * alpha), (1.0 / (eta + 1.0)));
	}
	else
	{
		betaq = std::pow((1.0 / (2.0 - rand * alpha)), (1.0 / (eta + 1.0)));
	}
	return betaq;
}
// ----------------------------------------------------------------------
bool CSimulatedBinaryCrossover::operator()(CIndividual* child1,
	CIndividual* child2,
	const CIndividual& parent1,
	const CIndividual& parent2,
	double cr,
	double eta) const
{
	*child1 = parent1;
	*child2 = parent2;

	if (MathAux::random(0.0, 1.0) > cr) return false; // not crossovered

	CIndividual::TDecVec& c1 = child1->vars(), & c2 = child2->vars();
	const CIndividual::TDecVec& p1 = parent1.vars(), & p2 = parent2.vars();

	for (size_t i = 0; i < c1.size(); i += 1)
	{
		if (MathAux::random(0.0, 1.0) > 0.5) continue; // these two variables are not crossovered
		if (std::fabs(p1[i] - p2[i]) <= MathAux::EPS) continue; // two values are the same

		double y1 = std::min(p1[i], p2[i]),
			y2 = std::max(p1[i], p2[i]);

		double lb = CIndividual::TargetProblem().lower_bounds()[i],
			ub = CIndividual::TargetProblem().upper_bounds()[i];

		double rand = MathAux::random(0.0, 1.0);

		// child 1
		double beta = 1.0 + (2.0 * (y1 - lb) / (y2 - y1)),
			alpha = 2.0 - std::pow(beta, -(eta + 1.0));
		double betaq = get_betaq(rand, alpha, eta);

		c1[i] = 0.5 * ((y1 + y2) - betaq * (y2 - y1));

		// child 2
		beta = 1.0 + (2.0 * (ub - y2) / (y2 - y1));
		alpha = 2.0 - std::pow(beta, -(eta + 1.0));
		betaq = get_betaq(rand, alpha, eta);

		c2[i] = 0.5 * ((y1 + y2) + betaq * (y2 - y1));

		// boundary checking
		c1[i] = std::min(ub, std::max(lb, c1[i]));
		c2[i] = std::min(ub, std::max(lb, c2[i]));

		if (MathAux::random(0.0, 1.0) <= 0.5)
		{
			std::swap(c1[i], c2[i]);
		}
	}

	return true;
```

```cpp
}// CSimulatedBinaryCrossover


// #include "mutation.h"
// #include "individual.h"
// #include "math_aux.h"
// #include "base_problem.h"

#include <cstddef>
#include <algorithm>
using std::size_t;

// ----------------------------------------------------------------------
// The implementation was adapted from the code of function real_mutate_ind() in mutation.c in
// http://www.iitk.ac.in/kangal/codes/nsga2/nsga2-gnuplot-v1.1.6.tar.gz
//
// ref: http://www.slideshare.net/paskorn/simulated-binary-crossover-presentation#
// ----------------------------------------------------------------------
bool CPolynomialMutation::operator()(CIndividual* indv, double mr, double eta) const
{
    //int j;
    //double rnd, delta1, delta2, mut_pow, deltaq;
    //double y, yl, yu, val, xy;

    bool mutated = false;

    CIndividual::TDecVec& x = indv->vars();

    for (size_t i = 0; i < x.size(); i += 1)
    {
        if (MathAux::random(0.0, 1.0) <= mr)
        {
            mutated = true;

            double y = x[i],
                lb = CIndividual::TargetProblem().lower_bounds()[i],
                ub = CIndividual::TargetProblem().upper_bounds()[i];

            double delta1 = (y - lb) / (ub - lb),
                delta2 = (ub - y) / (ub - lb);

            double mut_pow = 1.0 / (eta + 1.0);

            double rnd = MathAux::random(0.0, 1.0), deltaq = 0.0;
            if (rnd <= 0.5)
            {
                double xy = 1.0 - delta1;
                double val = 2.0 * rnd + (1.0 - 2.0 * rnd) * (std::pow(xy, (eta + 1.0)));
                deltaq = std::pow(val, mut_pow) - 1.0;
            }
            else
            {
                double xy = 1.0 - delta2;
                double val = 2.0 * (1.0 - rnd) + 2.0 * (rnd - 0.5) * (std::pow(xy, (eta + 1.0)));
                deltaq = 1.0 - (std::pow(val, mut_pow));
            }

            y = y + deltaq * (ub - lb);
            y = std::min(ub, std::max(lb, y));

            x[i] = y;
        }
    }
```

```
    return mutated;
}// CPolynomialMutation


// #include "comparator.h"
// #include "nondominated_sort.h"
// #include "population.h"

using namespace std;

CNondominatedSort NondominatedSort(ParetoDominate);
// ------------------------------------------------------------------

std::pair< std::vector<size_t>,std::vector< CNondominatedSort::TFrontMembers > > CNondominatedSort::operator()(const
CPopulation &pop) const
{
	CNondominatedSort::TFronts fronts;
	size_t num_assigned_individuals = 0;
	size_t rank = 1;
	vector<size_t> indv_ranks(pop.size(), 0);

	while (num_assigned_individuals < pop.size())
	{
		CNondominatedSort::TFrontMembers cur_front;

		for (size_t i=0; i<pop.size(); i+=1)
		{
			if (indv_ranks[i] > 0) continue; // already assigned a rank

			bool be_dominated = false;
			for (size_t j=0; j<cur_front.size(); j+=1)
			{
				if ( dominate(pop[ cur_front[j] ], pop[i]) ) // i is dominated
				{
					be_dominated = true;
					break;
				}
				else if ( dominate(pop[i], pop[ cur_front[j] ]) ) // i dominates a member in the current front
				{
					cur_front.erase(cur_front.begin()+j);
					j -= 1;
				}
			}
			if (!be_dominated)
			{
				cur_front.push_back(i);
			}
		}

		for (size_t i=0; i<cur_front.size(); i+=1)
		{
			indv_ranks[ cur_front[i] ] = rank;
		}
		fronts.push_back(cur_front);
		num_assigned_individuals += cur_front.size();

		rank += 1;
	}
	return { indv_ranks,fronts };

}// CNondominatedSort::operator()
// ------------------------------------------------------------------
```

```
// #include "crowding_distance.h"
#include <iostream>
#include <limits>
#include <algorithm>

std::vector<double> CalculateCrowdingDistance(CPopulation& cur, CNondominatedSort::TFronts& fronts)
{
    const size_t num_objs = cur[0].objs().size();
    const size_t num_individuals = cur.size();
    const size_t num_fronts = fronts.size();

    std::vector<double> crowding_distance(num_individuals, 0.0);

    for (size_t f = 0; f < num_objs; ++f)
    {
        for (size_t i = 0; i < num_fronts; ++i)
        {
            const size_t front_size = fronts[i].size();
            if (front_size <= 2)
            {
                // Skip fronts with 2 or fewer individuals as their crowding distance will be infinity
                                                crowding_distance[fronts[i][0]]=std::numeric_limits<double>::infinity();
                                                if(front_size==2)crowding_distance[fronts[i][1]]=std::numeric_limits<double>::infinity();
                continue;
            }

            std::sort(fronts[i].begin(), fronts[i].end(), [&](size_t a, size_t b) {
                return cur[a].objs()[f] < cur[b].objs()[f];
                });

            crowding_distance[fronts[i][0]] = crowding_distance[fronts[i][front_size - 1]] = std::numeric_limits<double>::infinity();
            const double obj_range = cur[fronts[i][front_size - 1]].objs()[f] - cur[fronts[i][0]].objs()[f];

            for (size_t j = 1; j < front_size - 1; ++j)
            {
                const size_t indv_index = fronts[i][j];
                crowding_distance[indv_index] += (cur[fronts[i][j + 1]].objs()[f] - cur[fronts[i][j - 1]].objs()[f]) / obj_range;
            }
        }
    }
    return crowding_distance;
}

void SortBasedOnCrowdingDistance(CPopulation& cur, CNondominatedSort::TFronts& fronts)
{
    std::vector<double> crowding_distance = CalculateCrowdingDistance(cur, fronts);

    // Sort individuals within each front based on crowding distance
    for (size_t i = 0; i < fronts.size(); ++i)
    {
        CNondominatedSort::TFrontMembers& front = fronts[i];
        std::sort(front.begin(), front.end(), [&](size_t a, size_t b) {
         return crowding_distance[a] > crowding_distance[b]; // Sort in descending order of crowding distance
         });
    }

}
```

```
// #include "survivor_selection.h"
// #include "population.h"
// #include "math_aux.h"
// #include "nondominated_sort.h"
// #include "crowding_distance.h"

#include <limits>
#include <algorithm>

using namespace std;

// ----------------------------------------------------------------------
// SurvivorSelection():
// ----------------------------------------------------------------------
void SurvivorSelection(CPopulation *pnext, CPopulation *pcur,  size_t PopSize)
{
	CPopulation &cur = *pcur, &next = *pnext;
	next.clear();

	// ---------- Step 4 in Algorithm 1: non-dominated sorting ----------
	CNondominatedSort::TFronts fronts = NondominatedSort(cur).second;

	SortBasedOnCrowdingDistance(cur, fronts);


	for (size_t t = 0; t < fronts.size() - 1; t += 1)
	{
		for (size_t i = 0; i < fronts[t].size(); i += 1) {
		  if (next.size() >= PopSize) break;
		  next.push_back(cur[fronts[t][i]]);
		}
	}

}
// ----------------------------------------------------------------------



// #include "gnuplot_interface.h"
#include <iostream>
#include <sstream>

using namespace std;

// ------------------------------------------------------------
// Ref:
// http://user.frdm.info/ckhung/b/ma/gnuplot.php
// ------------------------------------------------------------

Gnuplot::Gnuplot()
{
	// with -persist option you will see the windows as your program ends
	//gnuplotpipe=_popen("gnuplot -persist","w");
	//without that option you will not see the window

	// because I choose the terminal to output files so I don't want to see the window

	gnuplotpipe = _popen("gnuplot", "w");

	if (!gnuplotpipe)
	{
		cerr << ("Gnuplot not found !");
```

```
	}
}
// --------------------------------------------------------
Gnuplot::~Gnuplot()
{
	fprintf(gnuplotpipe, "exit\n");
	_pclose(gnuplotpipe);
}
// --------------------------------------------------------
void Gnuplot::operator()(const string& command)
{
	fprintf(gnuplotpipe, "%s\n", command.c_str());
	fflush(gnuplotpipe);
	// flush is necessary, nothing gets plotted else
}
// --------------------------------------------------------
void Gnuplot::set_title(const std::string& title)
{
	ostringstream ss;
	ss << "set title \"" << title << "\"";
	operator()(ss.str());
}
// --------------------------------------------------------
void Gnuplot::plot(const std::string& fname, std::size_t x, std::size_t y)
{
	ostringstream ss;

	ss << "plot \"" << fname << "\" using " << x << ":" << y;
	operator()(ss.str());
}
// --------------------------------------------------------
void Gnuplot::splot(const std::string& fname, std::size_t x, std::size_t y, std::size_t z)
{
	ostringstream ss;

	ss << "splot \"" << fname << "\" using " << x << ":" << y << ":" << z;
	operator()(ss.str());
}
// --------------------------------------------------------



// #include "assignment_problem.h"
// #include "individual.h"
// #include "math_aux.h"

#include <cmath>
#include <vector>
#include<iostream>

using std::size_t;
using std::cos;

// ------------------------------------------------------------------
//			CProblemAssignment
// ------------------------------------------------------------------

CProblemAssignment::CProblemAssignment(size_t M, size_t k, const std::string& name, double lbs, double ubs) :
	BProblem(name),
	M_(M),
	k_(k)
{
	lbs_.resize( k_, lbs); // lower bound
```

```
		ubs_.resize( k_, ubs); // upper bound
}


bool CProblemAssignment5::Evaluate(CIndividual* indv) const
{
		CIndividual::TDecVec& x = indv->vars();
		CIndividual::TObjVec& f = indv->objs();

		if (x.size() != k_) return false; // #variables does not match

		f.resize(M_, 0);
		for (size_t i = 1; i < k_; ++i) {
		f[0] += -10 * exp(-0.2 * sqrt(MathAux::square(x[i - 1]) + MathAux::square(x[i])));
		}
		for (size_t i = 0; i < k_; ++i) {
		f[1] += pow(abs(x[i]), 0.8) + 5 * sin(pow(x[i], 3));
		}


		return true;
}


bool CProblemAssignment6::Evaluate(CIndividual* indv) const
{
		CIndividual::TDecVec& x = indv->vars();
		CIndividual::TObjVec& f = indv->objs();
		//std::cout << (x.size() == k_)<<'\n';
		if (x.size() != k_) return false; // #variables does not match

		f.resize(M_, 0);
		f[0] = MathAux::square(x[0]);
		f[1] = MathAux::square(x[0] - 2);

		return true;
}


// #include "log.h"
// #include "population.h"
// #include "gnuplot_interface.h"

#include <fstream>
#include<iostream>
using namespace std;

#define OUTPUT_DECISION_VECTOR

bool SaveToFile(const std::string& fname, const CPopulation& pop, ios_base::openmode mode)
{
		ofstream ofile(fname.c_str(), mode);
		if (!ofile) return false;

		for (size_t i = 0; i < pop.size(); i += 1)
		{
#ifdef OUTPUT_DECISION_VECTOR
				for (size_t j = 0; j < pop[i].vars().size(); j += 1)
				{
						ofile << pop[i].vars()[j] << ' ';
				}
#endif
```

```
			for (size_t f = 0; f < pop[i].objs().size(); f += 1)
			{
				ofile << pop[i].objs()[f] << ' ';
			}
			ofile << endl;
		}
		ofile << endl;
		return true;
}
// ---------------------------------------------------------------------
bool ShowPopulation(Gnuplot& gplot, const CPopulation& pop, const std::string& legend)
{
		if (!SaveToFile(legend, pop, ios_base::out)) return false;


		size_t n = 0;
#ifdef OUTPUT_DECISION_VECTOR
		n = pop[0].vars().size();
#endif

		if (pop[0].objs().size() == 2)
		{
			gplot.plot(legend, n + 1, n + 2);
			return true;
		}
		else if (pop[0].objs().size() == 3)
		{
			gplot.splot(legend, n + 1, n + 2, n + 3);
			return true;
		}
		else
			return false;
}

// #include "assignment_problem.h"
// #include "nsgaii.h"
// #include "population.h"
// #include "gnuplot_interface.h"
// #include "log.h"

#include <ctime>
#include <cstdlib>
#include <iostream>

// #include "individual.h"
// #include "math_aux.h"

using namespace std;

int main()
{
		CNSGAII nsgaii("nsgaii");
		CPopulation solutions;
		Gnuplot gplot;

		const size_t NumRuns = 10;

		BProblem* problem5 = new CProblemAssignment5(2, 3,-5,5);
		BProblem* problem6 = new CProblemAssignment6(2, 1,-10,10);

		BProblem* problem = problem5;

		for (size_t r = 0; r < NumRuns; r += 1)
```

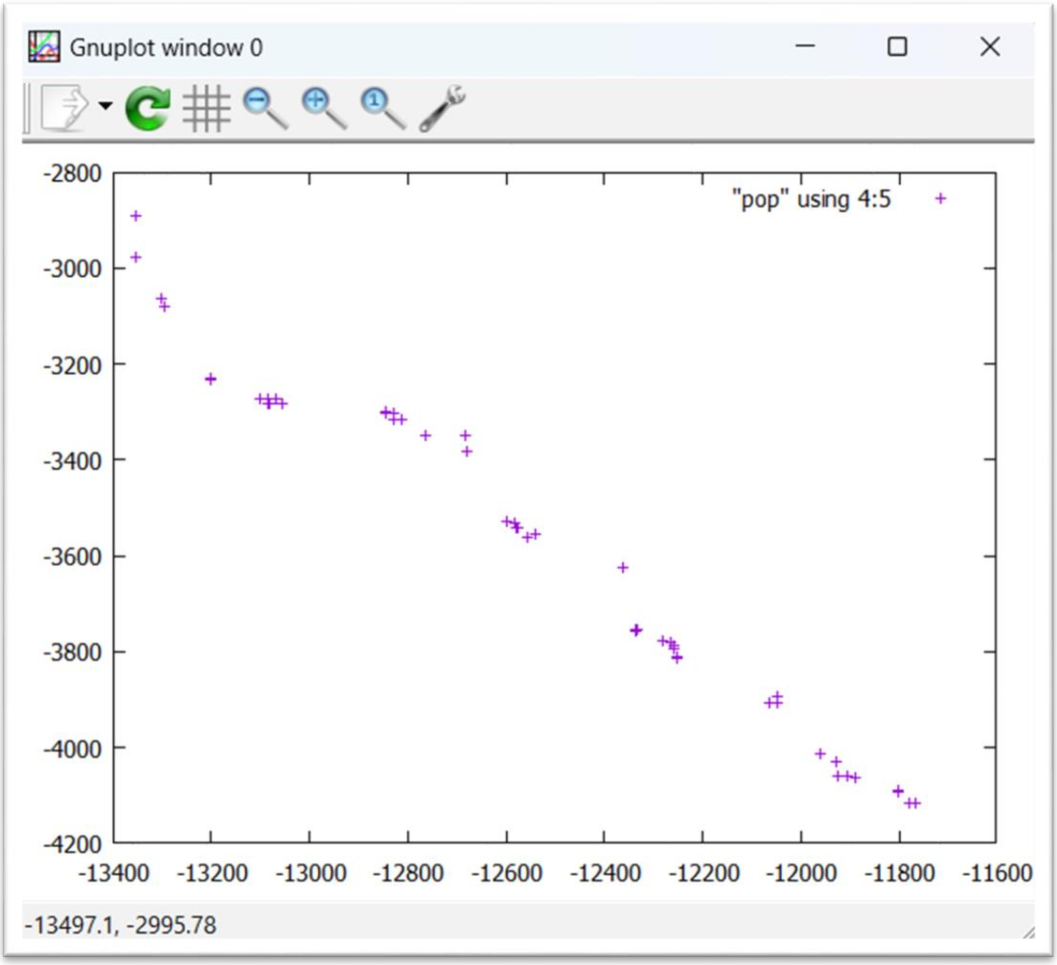
```
        {
                srand(r); cout << "Run Number: " << r << endl;

                nsgaii.Solve(&solutions, *problem);

                SaveToFile(nsgaii.name() + "-" + problem->name() + ".txt", solutions);
                ShowPopulation(gplot, solutions, "pop"); // system("pause");
        }
        delete problem;

        return 0;
}
```

Input: Terminal Input is not required Everything is Automated. If you want to change the parameters please update the main function and nsgaii function.

You can also add nsgaii.ini file and write parameter values.

Output:

Values of F1 and F2 after max iteration



The Values after the max iteration (x1, x2, x3, f1, f2)

```
1538   -1.11987 -0.138153 -0.9782 -11919.7 -4229.5
1539   -0.97126 -0.952095 -0.702945 -13717.2 -2538.64
1540   1.20987 0.00413672 -0.398633 -12945.1 -3714.37
1541   -1.08059 -0.997093 -0.664462 -13010 -3329.7
1542   -1.26487 -1.24351 -1.02411 -12586.5 -3917.05
1543   0.196504 -0.297933 0.718623 -12176.6 -3922.23
1544   -0.834474 -1.01118 -0.314952 -12605.8 -3911.48
1545   -1.22971 -1.1788 -1.19422 -13591.9 -2751.7
1546   -1.31946 -0.897347 -0.747646 -12858.4 -3804.83
1547   -0.928949 -0.537695 -0.987904 -13048.8 -3203.26
1548   -0.604696 -0.43047 0.460868 -13304.7 -2761.32
1549   -1.0798 -1.07967 0.619333 -11998.8 -4191.41
1550   -1.30669 -0.220159 0.223242 -13640.3 -2715.67
1551   -1.08799 -0.521459 -1.0103 -12039.6 -4047.66
1552   0.0108382 0.225293 -0.340068 -13260 -2900.69
1553   -1.06375 -1.11426 -0.0344402 -13234.6 -2979.45
1554   -0.769842 -0.421129 -0.254657 -13122.9 -3050.63
1555   -1.06451 -1.21176 -0.789812 -13302 -2770.91
1556   -1.07271 -1.20977 -0.715624 -13068.7 -3065.75
1557   -0.635114 -1.12646 -0.196319 -12120.3 -3934.08
1558   -1.0917 -1.12499 -0.394585 -13019.4 -3322.73
1559   -0.698961 -0.710698 0.39851 -12089.6 -4030.66
1560   -1.09635 -0.95656 -1.10018 -11936.9 -4219.55
1561   -1.08457 -1.33199 -0.7362 -13253.8 -2911.65
```

Hands On calculations:

Number of populations =4
Tournament parent selection
SBX crossover
Polynomial Mutation
Survivor selection Based on Crowding Distance and Rank

Generation Number =1

Parent selection

Sr. No.	x1	x2	x3	f1(x)	f2(x)
1	-4.9884	-2.64428	1.48152	-8.68714	13.3724
2	-4.25626	-2.29759	-1.39973	-9.63955	1.66223
3	-2.44713	4.8529	-1.81082	-6.92117	9.18446
4	4.34233	-3.21375	3.5757	-7.21749	9.56758

Crossover

Father	Mother			c1				c2	
			x1	x2	x3		x1	x2	x3
3	2	1	-4.23878	-2.28495	-1.81082	2	-2.46461	4.82254	-1.39973
3	3	3	-2.44713	4.8529	-1.81082	4	-2.44713	4.8529	-1.81082

Mutation

		Mutated			

		Value			
Sr. No.	x1	x2	x3	f1(x)	f2(x)
1	-4.23878	-2.28495	-1.81082	-16.32 17	17.1229
2	-1.80237	4.57723	-1.39973	-17.2175	13.0664
3	-2.44713	4.8529	-1.81082	-13.8423	18.3689
4	-1.70358	4.95464	-1.81082	-13.9098	26.3758

Survivor Selection

Sr. No.	x1	x2	x3	Rank	Crowding Distance	Next Generation
1	-4.9884	-2.64428	1.48152	2	1.92021	2
2	-4.25626	-2.29759	-1.39973	1	inf	6
3	-2.44713	4.8529	-1.81082	2	inf	3
4	4.34233	-3.21375	3.5757	2	0.715446	5
5	-4.23878	-2.28495	-1.81082	2	inf	
6	-1.80237	4.57723	-1.39973	1	inf	
7	-2.44713	4.8529	-1.81082	3	inf	
8	-1.70358	4.95464	-1.81082	3	inf	

Generation Number =2

Parent selection

Sr. No.	x1	x2	x3	f1(x)	f2(x)
1	-4.25626	-2.29759	-1.39973	-9.63955	1.66223
2	-1.80237	4.57723	-1.39973	-17.2175	13.0664
3	-2.44713	4.8529	-1.81082	-6.92117	9.18446
4	-4.23878	-2.28495	-1.81082	-16.32 17	17.1229

Crossover

Father	Mother			c1				c2	
			x1	x2	x3		x1	x2	x3
2	1	1	-1.80237	4.5803	-1.39973	2	-4.25626	-2.30231	-1.39973
1	3	3	-4.25626	4.8693	-1.39973	4	-2.44713	-2.34261	-1.81082

Mutation

		Mutated Value			
Sr. No.	x1	x2	x3	f1(x)	f2(x)
1	-1.80237	4.5803	-1.39973	-24.7911	24.3035
2	-4.25626	-2.30231	-2.28626	-18.6648	8.44429
3	-4.25626	4.93193	-1.39973	-15.9437	5.5974

4	-2.44713	-2.33324	-1.81082	-17.5459	11.4784

Survivor Selection

Sr. No.	x1	x2	x3	Rank	Crowding Distance	Next Generation
1	-4.25626	-2.29759	-1.39973	1	inf	1
2	-1.80237	4.57723	-1.39973	3	inf	5
3	-2.44713	4.8529	-1.81082	2	inf	6
4	-4.23878	-2.28495	-1.81082	4	inf	7
5	-1.80237	4.5803	-1.39973	1	inf	
6	-4.25626	-2.30231	-2.28626	1	1.4101	
7	-4.25626	4.93193	-1.39973	1	0.89520	
8	-2.44713	-2.33324	-1.81082	2	inf	

Best Answer: x = [-4.25626 -2.29759 -1.39973] (Randomly selected one from pareto-optimal front)

Problem 6:

Functions	Constraints	Search Domain
$\text{Minimize} = \begin{cases} f_1(x) = x^2 \\ f_2(x) = (x-2)^2 \end{cases}$		$-A \le x \le A$. Values of A from 10 to 10^5 have been used successfully. Higher values of A increase the difficulty of the problem.

C++ code for the SGA:

```
#ifndef MATH_AUX__
#define MATH_AUX__

#include <cstdlib>
#include <vector>

namespace MathAux
{
	const double PI = 3.1415926;
	const double EPS = 1.0e-14; // follow nsga-ii source code
	inline double square(double n) { return n * n; }
	inline double random(double lb, double ub) { return lb + (static_cast<double>(std::rand()) / RAND_MAX) * (ub - lb); }

	// ASF(): achievement scalarization function
	double ASF(const std::vector<double>& objs, const std::vector<double>& weight);
}

#endif
```

```cpp
#ifndef BASE_PROBLEM__
#define BASE_PROBLEM__

#include <string>
#include <vector>

// ---------------------------------------------------------------------
//                      BProblem: the base class of problems (e.g. ZDT and DTLZ)
// ---------------------------------------------------------------------
class CIndividual;

class BProblem
{
public:
	explicit BProblem(const std::string& name) :name_(name) {}
	virtual ~BProblem() {}

	virtual std::size_t num_variables() const = 0;
	virtual std::size_t num_objectives() const = 0;
	virtual bool Evaluate(CIndividual* indv) const = 0;

	const std::string& name() const { return name_; }
	const std::vector<double>& lower_bounds() const { return lbs_; }
	const std::vector<double>& upper_bounds() const { return ubs_; }

protected:
	std::string name_;

	std::vector<double> lbs_, // lower bounds of variables
		ubs_; // upper bounds of variables
};

#endif


#ifndef ASSIGNMENT_PROBLEM__
#define ASSIGNMENT_PROBLEM__

// #include "base_problem.h"
#include <cstddef>
#include<iostream>

// ---------------------------------------------------------------------
class CProblemAssignment : public BProblem
{
public:
	CProblemAssignment(std::size_t M, std::size_t k, const std::string& name, double lbs, double ubs);

	virtual std::size_t num_variables() const { return k_; }
	virtual std::size_t num_objectives() const { return M_; }

	virtual bool Evaluate(CIndividual* indv) const = 0;

protected:
	std::size_t M_; // number of objectives
	std::size_t k_; // number of variables
};


// ---------------------------------------------------------------------
//                      CProblemAssignment5
// ---------------------------------------------------------------------
```

```
class CProblemAssignment5 : public CProblemAssignment
{
public:
	explicit CProblemAssignment5(std::size_t M, std::size_t k, double lbs, double ubs) :CProblemAssignment(M, k, "Assignment5", 
lbs, ubs) {}
	virtual bool Evaluate(CIndividual* indv) const;
};


// --------------------------------------------------------------------
//			CProblemAssignment6
// --------------------------------------------------------------------

class CProblemAssignment6 : public CProblemAssignment
{
public:
	explicit CProblemAssignment6(std::size_t M, std::size_t k, double lbs, double ubs) :CProblemAssignment(M, k, "Assignment6", 
lbs, ubs) {
		//std::cout << "what happend !" << lbs << '\n';
	}
	virtual bool Evaluate(CIndividual* indv) const;
};

#endif

#ifndef COMPARATOR__
#define COMPARATOR__

class CIndividual;

// ---------------------------------------------------------------------------------
//				BComparator : the base class of comparison operators
// ---------------------------------------------------------------------------------
class BComparator
{
public:
	virtual ~BComparator() {}

	virtual bool		operator()(const CIndividual& l, const CIndividual& r) const = 0;
};



// ---------------------------------------------------------------------------------
//				CParetoDominate
// ---------------------------------------------------------------------------------

class CParetoDominate : public BComparator
{
public:
	virtual	bool	operator()(const CIndividual& l, const CIndividual& r) const;
};


extern CParetoDominate ParetoDominate;
#endif


#ifndef CROSSOVER__
#define CROSSOVER__

// ---------------------------------------------------------------------------------
```

```
//                    CSimulatedBinaryCrossover : simulated binary crossover (SBX)
// ---------------------------------------------------------------------------------


class CIndividual;
class CSimulatedBinaryCrossover
{
public:
	explicit CSimulatedBinaryCrossover(double cr = 1.0, double eta = 30) :cr_(cr), eta_(eta) {} // NSGA-III (t-EC 2014) setting

	void SetCrossoverRate(double cr) { cr_ = cr; }
	double CrossoverRate() const { return cr_; }
	void SetDistributionIndex(double eta) { eta_ = eta; }
	double DistributionIndex() const { return eta_; }

	bool operator()(CIndividual* c1, CIndividual* c2, const CIndividual& p1, const CIndividual& p2, double cr, double eta) const;
	bool operator()(CIndividual* c1, CIndividual* c2, const CIndividual& p1, const CIndividual& p2) const
	{
		return operator()(c1, c2, p1, p2, cr_, eta_);
	}

private:

	double get_betaq(double rand, double alpha, double eta) const;

	double cr_; // crossover rate
	double eta_; // distribution index
};


#endif

#include <string>
#include <stdio.h>

#include <string>
#include <cstddef>

// ---------------------------------------------------------------------
// Gnuplot
//
// This is just a very simple interface to call gnuplot in the program.
// Now it seems to work only under windows + visual studio.
// ---------------------------------------------------------------------

class Gnuplot
{
public:
	Gnuplot();
	~Gnuplot();

	// prohibit copying (VS2012 does not support 'delete')
	Gnuplot(const Gnuplot&);
	Gnuplot& operator=(const Gnuplot&);

	// send any command to gnuplot
	void operator ()(const std::string& command);

	void reset() { operator()("reset"); }
	void replot() { operator()("replot"); }
	void set_title(const std::string& title);

	void plot(const std::string& fname, std::size_t x, std::size_t y);
```

```cpp
	void splot(const std::string& fname, std::size_t x, std::size_t y, std::size_t z);

protected:
	FILE* gnuplotpipe;
};

#ifndef INDIVIDUAL__
#define INDIVIDUAL__

#include <vector>
#include <ostream>

// ---------------------------------------------------------------------
//			CIndividual
// ---------------------------------------------------------------------

class BProblem;

class CIndividual
{
public:
	typedef double TGene;
	typedef std::vector<TGene> TDecVec;
	typedef std::vector<double> TObjVec;

	explicit CIndividual(std::size_t num_vars = 0, std::size_t num_objs = 0);

	TDecVec& vars() { return variables_; }
	const TDecVec& vars() const { return variables_; }

	TObjVec& objs() { return objectives_; }
	const TObjVec& objs() const { return objectives_; }

	TObjVec& conv_objs() { return converted_objectives_; }
	const TObjVec& conv_objs() const { return converted_objectives_; }

	// if a target problem is set, memory will be allocated accordingly in the constructor
	static void SetTargetProblem(const BProblem& p) { target_problem_ = &p; }
	static const BProblem& TargetProblem();


private:
	TDecVec variables_;
	TObjVec objectives_;
	TObjVec converted_objectives_;

	static const BProblem* target_problem_;
};



std::ostream& operator << (std::ostream& os, const CIndividual& indv);

#endif

#ifndef INITIALIZATION__
#define INITIALIZATION__


// ---------------------------------------------------------------------
//			CRandomInitialization
// ---------------------------------------------------------------------
```

```cpp
class CIndividual;
class CPopulation;
class BProblem;

class CRandomInitialization
{
public:
	void operator()(CPopulation* pop, const BProblem& prob) const;
	void operator()(CIndividual* indv, const BProblem& prob) const;
};

extern CRandomInitialization RandomInitialization;

#endif

#ifndef POPULATION__
#define POPULATION__

// #include "individual.h"

#include <vector>

class CPopulation
{
public:
	explicit CPopulation(std::size_t s = 0) :individuals_(s) {}

	CIndividual& operator[](std::size_t i) { return individuals_[i]; }
	const CIndividual& operator[](std::size_t i) const { return individuals_[i]; }

	std::size_t size() const { return individuals_.size(); }
	void resize(size_t t) { individuals_.resize(t); }
	void push_back(const CIndividual& indv) { individuals_.push_back(indv); }
	void clear() { individuals_.clear(); }

private:
	std::vector<CIndividual> individuals_;
};

#endif

#ifndef ENVIRONMENTAL_SELECTION__
#define ENVIRONMENTAL_SELECTION__
#include <vector>


// ----------------------------------------------------------------------
//		The environmental selection mechanism is the key innovation of
//  the GA algorithm.
//
//  Check Algorithm I in the original paper of NSGA-III.
// ----------------------------------------------------------------------

class CPopulation;
class CReferencePoint;

void SurvivorSelection(CPopulation* pnext, // population in the next generation
	CPopulation* pcur,  // population in the current generation
	size_t PopSize);

#endif

#ifndef LOG__
```

```cpp
#define LOG__

#include <string>
#include <fstream>  // Include <fstream> for std::ios_base and std::ios_base::openmode

class CPopulation;
class Gnuplot;

// Save a population into the designated file.
bool SaveToFile(const std::string& fname, const CPopulation& pop, std::ios_base::openmode mode = std::ios_base::app);

// Show a population by calling gnuplot.
bool ShowPopulation(Gnuplot& gplot, const CPopulation&, const std::string& legend = "pop");

#endif

#ifndef MUTATION__
#define MUTATION__

// ----------------------------------------------------------------------------------
//                      CPolynomialMutation : polynomial mutation
// ----------------------------------------------------------------------------------

class CIndividual;

class CPolynomialMutation
{
public:
	explicit CPolynomialMutation(double mr = 0.0, double eta = 20) :mr_(mr), eta_(eta) {}

	void SetMutationRate(double mr) { mr_ = mr; }
	double MutationRate() const { return mr_; }
	void SetDistributionIndex(double eta) { eta_ = eta; }
	double DistributionIndex() const { return eta_; }

	bool operator()(CIndividual* c, double mr, double eta) const;
	bool operator()(CIndividual* c) const
	{
		return operator()(c, mr_, eta_);
	}

private:
	double mr_, // mutation rate
		eta_; // distribution index
};

#endif

#ifndef GA__
#define GA__

#include <cstddef>
#include <string>

// ----------------------------------------------------------------------------------

class BProblem;
class CPopulation;

class CGA
{
public:
	explicit CGA(const std::string& inifile_name = "");
```

```
	void Solve(CPopulation* solutions, const BProblem& prob);

	const std::string& name() const { return name_; }
private:
	std::string name_;
	std::size_t obj_division_p_;
	std::size_t gen_num_;
	double  pc_, // crossover rate
		pm_, // mutation rate
		eta_c_, // eta in SBX
		eta_m_; // eta in Polynomial Mutation
};


#endif

// #include "ga.h"
// #include "base_problem.h"
// #include "individual.h"
// #include "population.h"

// #include "initialization.h"
// #include "crossover.h"
// #include "mutation.h"
// #include "survivor_selection.h"

// #include "gnuplot_interface.h"
// #include "log.h"
#include "windows.h" // for Sleep()

#include <vector>
#include <fstream>
#include<iostream>

using namespace std;

CGA::CGA(const string& inifile_name) :
	name_("GA"),
	obj_division_p_(12),
	gen_num_(1000),
	pc_(1.0), // default setting in GA
	eta_c_(30), // default setting
	eta_m_(20) // default setting
{
	if (inifile_name == "") return;

	ifstream inifile(inifile_name);
	if (!inifile) return;

	string dummy;
	inifile >> dummy >> dummy >> name_;
	inifile >> dummy >> dummy >> obj_division_p_;
	inifile >> dummy >> dummy >> gen_num_;
	inifile >> dummy >> dummy >> pc_;
	inifile >> dummy >> dummy >> eta_c_;
	inifile >> dummy >> dummy >> eta_m_;
}
// ----------------------------------------------------------------------
void CGA::Solve(CPopulation* solutions, const BProblem& problem)
{
	CIndividual::SetTargetProblem(problem);
```

```
	size_t PopSize = 50;
	while (PopSize % 4) PopSize += 1;

	//CPopulation pop[2] = { CPopulation(PopSize) };
	std::vector<CPopulation> pop(2, CPopulation(PopSize));
	CSimulatedBinaryCrossover SBX(pc_, eta_c_);
	CPolynomialMutation PolyMut(1.0 / problem.num_variables(), eta_m_);

	Gnuplot gplot;

	int cur = 0, next = 1;

	//std::cout << problem.lower_bounds()[0] << '\n';
	RandomInitialization(&pop[cur], problem);
	for (size_t i = 0; i < PopSize; i += 1)
	{
		problem.Evaluate(&pop[cur][i]);
	}

	for (size_t t = 0; t < gen_num_; t += 1)
	{
		pop[cur].resize(PopSize * 2);

		for (size_t i = 0; i < PopSize; i += 2)
		{
			int father = rand() % PopSize,
				mother = rand() % PopSize;

			SBX(&pop[cur][PopSize + i], &pop[cur][PopSize + i + 1], pop[cur][father], pop[cur][mother]);

			PolyMut(&pop[cur][PopSize + i]);
			PolyMut(&pop[cur][PopSize + i + 1]);

			problem.Evaluate(&pop[cur][PopSize + i]);
			problem.Evaluate(&pop[cur][PopSize + i + 1]);
		}

		SurvivorSelection(&pop[next], &pop[cur], PopSize);


		//ShowPopulation(gplot, pop[next], "pop"); Sleep(10);

		std::swap(cur, next);
	}

	*solutions = pop[cur];
}


// #include "comparator.h"
// #include "individual.h"

// ------------------------------------------------------------------------------

CParetoDominate ParetoDominate;

// ------------------------------------------------------------------------------
//						CParetoDominate
// ------------------------------------------------------------------------------
bool CParetoDominate::operator()(const CIndividual& l, const CIndividual& r) const
{
	bool better = false;
	for (size_t f = 0; f < l.objs().size(); f += 1)
```

```
		{
			if (l.objs()[f] > r.objs()[f])
				return false;
			else if (l.objs()[f] < r.objs()[f])
				better = true;
		}

		return better;

}// CParetoDominate::operator()
// ------------------------------------------------------------------------------

#include <vector>
// #include "math_aux.h"
#include<limits>

using namespace std;

namespace MathAux
{

// --------------------------------------------------------------------
// ASF: Achivement Scalarization Function
// --------------------------------------------------------------------
double ASF(const vector<double> &objs, const vector<double> &weight)
{
		double max_ratio = -numeric_limits<double>::max();
		for (size_t f=0; f<objs.size(); f+=1)
		{
			double w = weight[f]?weight[f]:0.00001;
			max_ratio = std::max(max_ratio, objs[f]/w);
		}
		return max_ratio;
}
// -------------------------------------------------------------------




}// namespace MathAux



// #include "individual.h"
// #include "base_problem.h"

using std::size_t;


const BProblem* CIndividual::target_problem_ = 0;
// --------------------------------------------------------------------
CIndividual::CIndividual(std::size_t num_vars, std::size_t num_objs) :
		variables_(num_vars),
		objectives_(num_objs),
		converted_objectives_(num_objs)
{
		if (target_problem_ != 0)
		{
			variables_.resize(target_problem_->num_variables());
			objectives_.resize(target_problem_->num_objectives());
			converted_objectives_.resize(target_problem_->num_objectives());
		}
}
```

```cpp
// ---------------------------------------------------------------------
const BProblem& CIndividual::TargetProblem() { return *target_problem_; }
// ---------------------------------------------------------------------
std::ostream& operator << (std::ostream& os, const CIndividual& indv)
{
	for (size_t i = 0; i < indv.vars().size(); i += 1)
	{
		os << indv.vars()[i] << ' ';
	}

	os << " => ";
	for (size_t f = 0; f < indv.objs().size(); f += 1)
	{
		os << indv.objs()[f] << ' ';
	}

	return os;
}

// ---------------------------------------------------------------------

// #include "initialization.h"
// #include "base_problem.h"
// #include "individual.h"
// #include "population.h"
// #include "math_aux.h"

#include <cstddef>
#include<iostream>
using std::size_t;

CRandomInitialization RandomInitialization;

void CRandomInitialization::operator()(CIndividual* indv, const BProblem& prob) const
{
	CIndividual::TDecVec& x = indv->vars();
	x.resize(prob.num_variables());
	for (size_t i = 0; i < x.size(); i += 1)
	{
		x[i] = MathAux::random(prob.lower_bounds()[i], prob.upper_bounds()[i]);
	}

}
// ---------------------------------------------------------------------
void CRandomInitialization::operator()(CPopulation* pop, const BProblem& prob) const
{
	for (size_t i = 0; i < pop->size(); i += 1)
	{
		(*this)(&(*pop)[i], prob);
	}
}



// #include "crossover.h"
// #include "individual.h"
// #include "math_aux.h"
// #include "base_problem.h"

#include <cmath>
#include <algorithm>
#include <cstddef>
using std::size_t;
```

```
// ----------------------------------------------------------------------
// The implementation was adapted from the code of function realcross() in crossover.c
// http://www.iitk.ac.in/kangal/codes/nsga2/nsga2-gnuplot-v1.1.6.tar.gz
//
// ref: http://www.slideshare.net/paskorn/simulated-binary-crossover-presentation#
// ----------------------------------------------------------------------
double CSimulatedBinaryCrossover::get_betaq(double rand, double alpha, double eta) const
{
	double betaq = 0.0;
	if (rand <= (1.0 / alpha))
	{
		betaq = std::pow((rand * alpha), (1.0 / (eta + 1.0)));
	}
	else
	{
		betaq = std::pow((1.0 / (2.0 - rand * alpha)), (1.0 / (eta + 1.0)));
	}
	return betaq;
}
// ----------------------------------------------------------------------
bool CSimulatedBinaryCrossover::operator()(CIndividual* child1,
	CIndividual* child2,
	const CIndividual& parent1,
	const CIndividual& parent2,
	double cr,
	double eta) const
{
	*child1 = parent1;
	*child2 = parent2;

	if (MathAux::random(0.0, 1.0) > cr) return false; // not crossovered

	CIndividual::TDecVec& c1 = child1->vars(), & c2 = child2->vars();
	const CIndividual::TDecVec& p1 = parent1.vars(), & p2 = parent2.vars();

	for (size_t i = 0; i < c1.size(); i += 1)
	{
		if (MathAux::random(0.0, 1.0) > 0.5) continue; // these two variables are not crossovered
		if (std::fabs(p1[i] - p2[i]) <= MathAux::EPS) continue; // two values are the same

		double y1 = std::min(p1[i], p2[i]),
			y2 = std::max(p1[i], p2[i]);

		double lb = CIndividual::TargetProblem().lower_bounds()[i],
			ub = CIndividual::TargetProblem().upper_bounds()[i];

		double rand = MathAux::random(0.0, 1.0);

		// child 1
		double beta = 1.0 + (2.0 * (y1 - lb) / (y2 - y1)),
			alpha = 2.0 - std::pow(beta, -(eta + 1.0));
		double betaq = get_betaq(rand, alpha, eta);

		c1[i] = 0.5 * ((y1 + y2) - betaq * (y2 - y1));

		// child 2
		beta = 1.0 + (2.0 * (ub - y2) / (y2 - y1));
		alpha = 2.0 - std::pow(beta, -(eta + 1.0));
		betaq = get_betaq(rand, alpha, eta);

		c2[i] = 0.5 * ((y1 + y2) + betaq * (y2 - y1));
```

```
            // boundary checking
            c1[i] = std::min(ub, std::max(lb, c1[i]));
            c2[i] = std::min(ub, std::max(lb, c2[i]));

            if (MathAux::random(0.0, 1.0) <= 0.5)
            {
                std::swap(c1[i], c2[i]);
            }
        }

        return true;
}// CSimulatedBinaryCrossover



// #include "mutation.h"
// #include "individual.h"
// #include "math_aux.h"
// #include "base_problem.h"

#include <cstddef>
#include <algorithm>
using std::size_t;

// ----------------------------------------------------------------------
// The implementation was adapted from the code of function real_mutate_ind() in mutation.c in
// http://www.iitk.ac.in/kangal/codes/nsga2/nsga2-gnuplot-v1.1.6.tar.gz
//
// ref: http://www.slideshare.net/paskorn/simulated-binary-crossover-presentation#
// ---------------------------------------------------------------------
bool CPolynomialMutation::operator()(CIndividual* indv, double mr, double eta) const
{
    //int j;
    //double rnd, delta1, delta2, mut_pow, deltaq;
    //double y, yl, yu, val, xy;

    bool mutated = false;

    CIndividual::TDecVec& x = indv->vars();

    for (size_t i = 0; i < x.size(); i += 1)
    {
        if (MathAux::random(0.0, 1.0) <= mr)
        {
            mutated = true;

            double y = x[i],
                lb = CIndividual::TargetProblem().lower_bounds()[i],
                ub = CIndividual::TargetProblem().upper_bounds()[i];

            double delta1 = (y - lb) / (ub - lb),
                delta2 = (ub - y) / (ub - lb);

            double mut_pow = 1.0 / (eta + 1.0);

            double rnd = MathAux::random(0.0, 1.0), deltaq = 0.0;
            if (rnd <= 0.5)
            {
                double xy = 1.0 - delta1;
                double val = 2.0 * rnd + (1.0 - 2.0 * rnd) * (pow(xy, (eta + 1.0)));
                deltaq = pow(val, mut_pow) - 1.0;
            }
            else
```

```cpp
        {
            double xy = 1.0 - delta2;
            double val = 2.0 * (1.0 - rnd) + 2.0 * (rnd - 0.5) * (pow(xy, (eta + 1.0)));
            deltaq = 1.0 - (pow(val, mut_pow));
        }

        y = y + deltaq * (ub - lb);
        y = std::min(ub, std::max(lb, y));

        x[i] = y;
      }
   }

   return mutated;
}// CPolynomialMutation

// #include "survivor_selection.h"
// #include "population.h"
// #include "math_aux.h"

#include <limits>
#include <algorithm>

using namespace std;

// ----------------------------------------------------------------------
// SurvivorSelection():
// ----------------------------------------------------------------------
void SurvivorSelection(CPopulation *pnext, CPopulation *pcur,  size_t PopSize)
{
		CPopulation &cur = *pcur, &next = *pnext;
		next.clear();
		std::vector<size_t> index;
		for (int i = 0; i < cur.size(); i++)index.push_back(i);


		std::sort(index.begin(), index.end(), [&](size_t a, size_t b) {
				return cur[a].objs()[0]+ cur[a].objs()[1] < cur[b].objs()[0]+ cur[b].objs()[1];
				});


		for (size_t t = 0; t < PopSize ; t += 1)
		{
				 next.push_back(cur[index[t]]);
		}

}
// ----------------------------------------------------------------------



// #include "gnuplot_interface.h"
#include <iostream>
#include <sstream>

using namespace std;

// ---------------------------------------------------------
// Ref:
// http://user.frdm.info/ckhung/b/ma/gnuplot.php
// ---------------------------------------------------------

Gnuplot::Gnuplot()
```

```
{
	// with -persist option you will see the windows as your program ends
	//gnuplotpipe=_popen("gnuplot -persist","w");
	//without that option you will not see the window

	// because I choose the terminal to output files so I don't want to see the window

	gnuplotpipe = _popen("gnuplot", "w");

	if (!gnuplotpipe)
	{
		cerr << ("Gnuplot not found !");
	}
}
// ----------------------------------------------------------
Gnuplot::~Gnuplot()
{
	fprintf(gnuplotpipe, "exit\n");
	_pclose(gnuplotpipe);
}
// ----------------------------------------------------------
void Gnuplot::operator()(const string& command)
{
	fprintf(gnuplotpipe, "%s\n", command.c_str());
	fflush(gnuplotpipe);
	// flush is necessary, nothing gets plotted else
}
// ----------------------------------------------------------
void Gnuplot::set_title(const std::string& title)
{
	ostringstream ss;
	ss << "set title \"" << title << "\"";
	operator()(ss.str());
}
// ----------------------------------------------------------
void Gnuplot::plot(const std::string& fname, std::size_t x, std::size_t y)
{
	ostringstream ss;

	ss << "plot \"" << fname << "\" using " << x << ":" << y;
	operator()(ss.str());
}
// ----------------------------------------------------------
void Gnuplot::splot(const std::string& fname, std::size_t x, std::size_t y, std::size_t z)
{
	ostringstream ss;

	ss << "splot \"" << fname << "\" using " << x << ":" << y << ":" << z;
	operator()(ss.str());
}
// ----------------------------------------------------------


// #include "assignment_problem.h"
// #include "individual.h"
// #include "math_aux.h"

#include <cmath>
#include <vector>
#include<iostream>

using std::size_t;
using std::cos;
```

```cpp
// ----------------------------------------------------------------------
//                        CProblemAssignment
// ----------------------------------------------------------------------

CProblemAssignment::CProblemAssignment(size_t M, size_t k, const std::string& name, double lbs, double ubs) :
            BProblem(name),
            M_(M),
            k_(k)
{
            lbs_.resize( k_, lbs); // lower bound
            ubs_.resize( k_, ubs); // upper bound
}


bool CProblemAssignment5::Evaluate(CIndividual* indv) const
{
            CIndividual::TDecVec& x = indv->vars();
            CIndividual::TObjVec& f = indv->objs();

            if (x.size() != k_) return false; // #variables does not match

            f.resize(M_, 0);
            for (size_t i = 1; i < k_; ++i) {
            f[0] += -10 * exp(-0.2 * sqrt(MathAux::square(x[i - 1]) + MathAux::square(x[i])));
            }
            for (size_t i = 0; i < k_; ++i) {
            f[1] += pow(abs(x[i]), 0.8) + 5 * sin(pow(x[i], 3));
            }


            return true;
}


bool CProblemAssignment6::Evaluate(CIndividual* indv) const
{
            CIndividual::TDecVec& x = indv->vars();
            CIndividual::TObjVec& f = indv->objs();
            //std::cout << (x.size() == k_)<<'\n';
            if (x.size() != k_) return false; // #variables does not match

            f.resize(M_, 0);
            f[0] = MathAux::square(x[0]);
            f[1] = MathAux::square(x[0] - 2);

            return true;
}



// #include "log.h"
// #include "population.h"
// #include "gnuplot_interface.h"

#include <fstream>
#include<iostream>
using namespace std;

#define OUTPUT_DECISION_VECTOR

bool SaveToFile(const std::string& fname, const CPopulation& pop, ios_base::openmode mode)
{
```

```cpp
	ofstream ofile(fname.c_str(), mode);
	if (!ofile) return false;

	for (size_t i = 0; i < pop.size(); i += 1)
	{
#ifdef OUTPUT_DECISION_VECTOR

		for (size_t j = 0; j < pop[i].vars().size(); j += 1)
		{
			ofile << pop[i].vars()[j] << ' ';
		}
#endif
		//std::cout << pop[i].objs().size() << endl;
		for (size_t f = 0; f < pop[i].objs().size(); f += 1)
		{
			ofile << pop[i].objs()[f] << ' ';
		}

	   ofile << pop[i].objs()[0] + pop[i].objs()[1] << ' ';

	   ofile << endl;
	}
	ofile << endl;
	return true;
}
// ---------------------------------------------------------------------
bool ShowPopulation(Gnuplot& gplot, const CPopulation& pop, const std::string& legend)
{
	if (!SaveToFile(legend, pop, ios_base::out)) return false;


	size_t n = 0;
#ifdef OUTPUT_DECISION_VECTOR
	n = pop[0].vars().size();
#endif

	if (pop[0].objs().size() == 2)
	{
		gplot.plot(legend, n + 1, n + 2);
		return true;
	}
	else if (pop[0].objs().size() == 3)
	{
		gplot.splot(legend, n + 1, n + 2, n + 3);
		return true;
	}
	else
		return false;
}


// #include "assignment_problem.h"
// #include "ga.h"
// #include "population.h"
// #include "gnuplot_interface.h"
// #include "log.h"

#include <ctime>
#include <cstdlib>
#include <iostream>

// #include "individual.h"
// #include "math_aux.h"
```

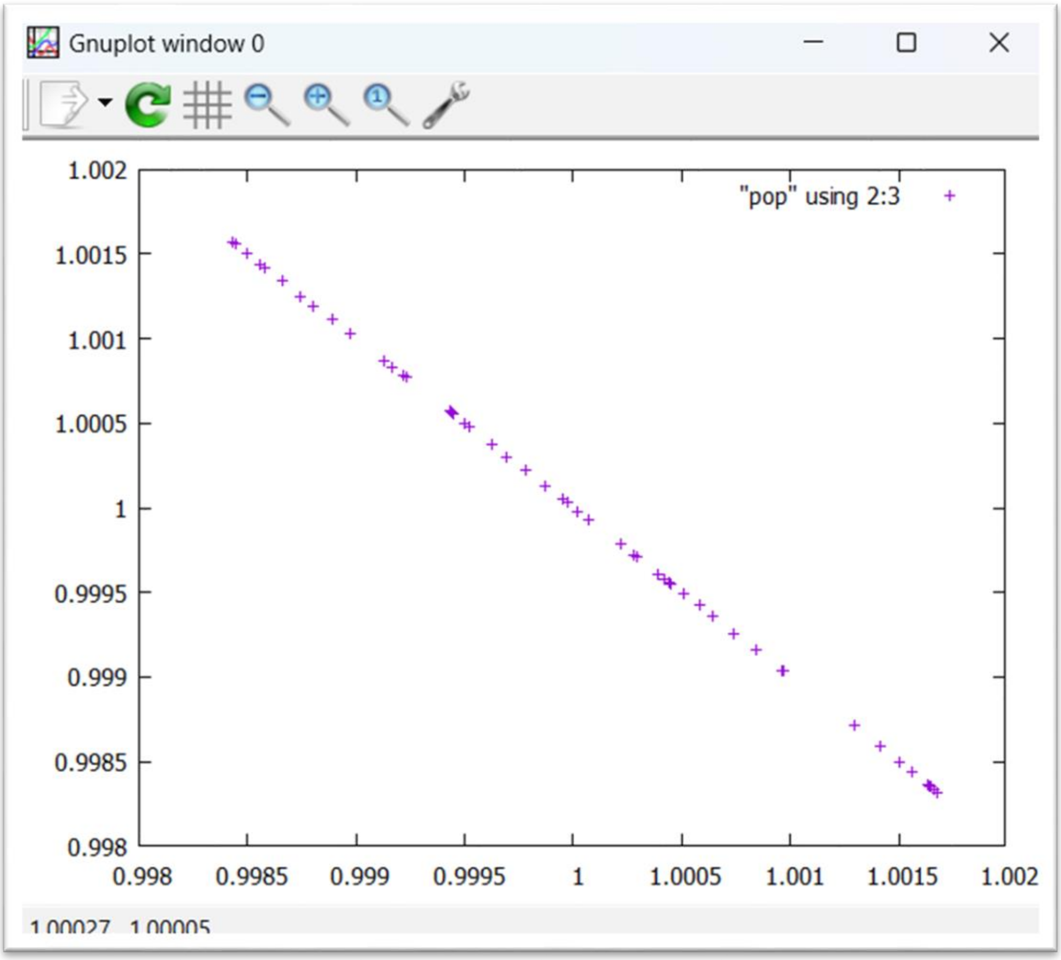
```
using namespace std;

int main()
{
	CGA ga("ga");
	CPopulation solutions;
	Gnuplot gplot;

	const size_t NumRuns = 10;

	BProblem* problem5 = new CProblemAssignment5(2, 3,-5,5);
	BProblem* problem6 = new CProblemAssignment6(2, 1,-10,10);

	BProblem* problem = problem6;

	//std::cout << problem->upper_bounds()[0] << '\n';

	for (size_t r = 0; r < NumRuns; r += 1)
	{
		srand(r); cout << "Run Number: " << r << endl;

		ga.Solve(&solutions, *problem);
		SaveToFile(ga.name() + "-" + problem->name() + ".txt", solutions);
		ShowPopulation(gplot, solutions, "pop"); system("pause");
	}
	delete problem;

	return 0;
}
```

Input: Terminal Input is not required Everything is Automated. If you want to change the parameters please update the main function and ga function.

You can also add ga.ini file and write parameter values.

Output:

Values of F1 and F2 after Max iteration



The Values after the max iteration (x, f1, f2, f1+f2)

```
0.999989 0.999977 1.00002 2
1.00003 1.00007 0.999934 2
0.999953 0.999906 1.00009 2
0.999924 0.999849 1.00015 2
1.00008 1.00015 0.999848 2
1.00011 1.00021 0.999789 2
0.999891 0.999782 1.00022 2
0.999849 0.999699 1.0003 2
0.999811 0.999621 1.00038 2
0.999803 0.999607 1.00039 2
1.00021 1.00041 0.999588 2
1.00021 1.00042 0.999581 2
1.00026 1.00053 0.999474 2
1.00027 1.00054 0.999456 2
1.00032 1.00065 0.999351 2
1.00033 1.00067 0.999333 2
1.00038 1.00075 0.999249 2
0.999594 0.999187 1.00081 2
0.999575 0.999151 1.00085 2
0.999548 0.999097 1.0009 2
```

Hands On calculations:

Number of populations =4
Tournament parent selection
SBX crossover
Polynomial Mutation
Survivor selection Fitness propionate

Generation Number =1

Parent selection

Sr. No.	x1	f1(x)	f2(x)	g(x)
1	-9.97681	99.5367	143.444	242.981
2	-5.28855	27.9688	53.123	81.0918
3	2.96304	8.77962	0.92745	9.70707
4	-8.51253	72.4631	110.513	182.976

Crossover

Father	Mother		c1		c2
			x1		x1
4	2	1	-8.51253	2	-5.28855
3	2	3	-5.20879	4	2.88328

Mutation

	Mutated Value			
Sr. No.	x1	f1(x)	f2(x)	g(x)
1	-6.67101	44.5024	75.1864	119.689
2	-4.12765	17.0375	37.5481	54.5855
3	-5.31273	28.2251	53.476	81.7011
4	5.4853	30.0885	12.1473	42.2358

Survivor Selection

Sr. No.	x1	g(x)		Next Generation
	Old Population			
1	-9.97681	242.981	1	3
2	-5.28855	81.0918	2	8
3	2.96304	9.70707	3	6
4	-8.51253	182.976	4	2
	New Population			
5	-6.67101	119.689		
6	-4.12765	54.5855		
7	-5.31273	81.7011		
8	5.4853	42.2358		

Generation Number =2

Parent selection

Sr. No.	x1	f1(x)	f2(x)	g(x)
1	2.96304	8.77962	0.92745	9.70707
2	5.4853	30.0885	12.1473	42.2358
3	-4.12765	17.0375	37.5481	54.5855
4	-5.28855	27.9688	53.123	81.0918

Crossover

Father	Mother		c1		c2
			x1		x1
4	2	1	-4.10007	2	-5.31613
1	4	3	2.96304	4	-5.28855

Mutation

	Mutated Value			
Sr. No.	x1	f1(x)	f2(x)	g(x)
1	-1.0203	1.04102	9.12222	10.1632
2	-8.16728	66.7044	103.373	170.078
3	2.71579	7.37553	0.51236	7.88789
4	-7.84949	61.6146	97.0125	158.627

Survivor Selection

Sr. No.	x1	g(x)		Next Generation
	Old Population			
1	2.96304	9.70707	1	7
2	5.4853	42.2358	2	1
3	-4.12765	54.5855	3	5
4	-5.28855	81.0918	4	2
	New Population			
5	-1.0203	10.1632		
6	-8.16728	170.078		
7	2.71579	7.88789		
8	-7.84949	158.627		

Best Answer: x = [2.71579]

C++ code for the NSGAII:

```cpp
#ifndef MATH_AUX__
#define MATH_AUX__

#include <cstdlib>
#include <vector>

namespace MathAux
{
	const double PI = 3.1415926;
	const double EPS = 1.0e-14; // follow nsga-ii source code
	inline double square(double n) { return n * n; }
	inline double random(double lb, double ub) { return lb + (static_cast<double>(std::rand()) / RAND_MAX) * (ub - lb); }

	// ASF(): achievement scalarization function
	double ASF(const std::vector<double>& objs, const std::vector<double>& weight);
}

#endif


#ifndef BASE_PROBLEM__
#define BASE_PROBLEM__

#include <string>
#include <vector>

// ----------------------------------------------------------------------
//			BProblem: the base class of problems (e.g. ZDT and DTLZ)
// ----------------------------------------------------------------------
class CIndividual;

class BProblem
{
public:
	explicit BProblem(const std::string& name) :name_(name) {}
	virtual ~BProblem() {}

	virtual std::size_t num_variables() const = 0;
	virtual std::size_t num_objectives() const = 0;
	virtual bool Evaluate(CIndividual* indv) const = 0;

	const std::string& name() const { return name_; }
	const std::vector<double>& lower_bounds() const { return lbs_; }
	const std::vector<double>& upper_bounds() const { return ubs_; }

protected:
	std::string name_;

	std::vector<double> lbs_, // lower bounds of variables
			ubs_; // upper bounds of variables
};

#endif


#ifndef ASSIGNMENT_PROBLEM__
#define ASSIGNMENT_PROBLEM__

//#include "base_problem.h"
#include <cstddef>
#include<iostream>
```

```
// ----------------------------------------------------------------------
class CProblemAssignment : public BProblem
{
public:
	CProblemAssignment(std::size_t M, std::size_t k, const std::string& name, double lbs, double ubs);

	virtual std::size_t num_variables() const { return k_; }
	virtual std::size_t num_objectives() const { return M_; }

	virtual bool Evaluate(CIndividual* indv) const = 0;

protected:
	std::size_t M_; // number of objectives
	std::size_t k_; // number of variables
};


// ----------------------------------------------------------------------
//			CProblemAssignment5
// ----------------------------------------------------------------------

class CProblemAssignment5 : public CProblemAssignment
{
public:
	explicit CProblemAssignment5(std::size_t M, std::size_t k, double lbs, double ubs) :CProblemAssignment(M, k, "Assignment5",
lbs, ubs) {}
	virtual bool Evaluate(CIndividual* indv) const;
};


// ----------------------------------------------------------------------
//			CProblemAssignment6
// ----------------------------------------------------------------------

class CProblemAssignment6 : public CProblemAssignment
{
public:
	explicit CProblemAssignment6(std::size_t M, std::size_t k, double lbs, double ubs) :CProblemAssignment(M, k, "Assignment6",
lbs, ubs) {
		//std::cout << "what happend !" << lbs << '\n';
	}
	virtual bool Evaluate(CIndividual* indv) const;
};

#endif

#ifndef COMPARATOR__
#define COMPARATOR__

class CIndividual;

// ---------------------------------------------------------------------------------
//				BComparator : the base class of comparison operators
// ---------------------------------------------------------------------------------
class BComparator
{
public:
	virtual ~BComparator() {}

	virtual bool		operator()(const CIndividual& l, const CIndividual& r) const = 0;
};
```

```
// ---------------------------------------------------------------------------------
//                                  CParetoDominate
// ---------------------------------------------------------------------------------

class CParetoDominate : public BComparator
{
public:
	virtual	bool	operator()(const CIndividual& l, const CIndividual& r) const;
};


extern CParetoDominate ParetoDominate;
#endif


#ifndef CROSSOVER__
#define CROSSOVER__

// ---------------------------------------------------------------------------------
//                       CSimulatedBinaryCrossover : simulated binary crossover (SBX)
// ---------------------------------------------------------------------------------


class CIndividual;
class CSimulatedBinaryCrossover
{
public:
	explicit CSimulatedBinaryCrossover(double cr = 1.0, double eta = 30) :cr_(cr), eta_(eta) {} // NSGA-III (t-EC 2014) setting

	void SetCrossoverRate(double cr) { cr_ = cr; }
	double CrossoverRate() const { return cr_; }
	void SetDistributionIndex(double eta) { eta_ = eta; }
	double DistributionIndex() const { return eta_; }

	bool operator()(CIndividual* c1, CIndividual* c2, const CIndividual& p1, const CIndividual& p2, double cr, double eta) const;
	bool operator()(CIndividual* c1, CIndividual* c2, const CIndividual& p1, const CIndividual& p2) const
	{
		return operator()(c1, c2, p1, p2, cr_, eta_);
	}

private:

	double get_betaq(double rand, double alpha, double eta) const;

	double cr_; // crossover rate
	double eta_; // distribution index
};


#endif


#include <string>
#include <stdio.h>

#include <string>
#include <cstddef>

// ----------------------------------------------------------------------
// Gnuplot
//
// This is just a very simple interface to call gnuplot in the program.
```

```
// Now it seems to work only under windows + visual studio.
// --------------------------------------------------------------------

class Gnuplot
{
public:
	Gnuplot();
	~Gnuplot();

	// prohibit copying (VS2012 does not support 'delete')
	Gnuplot(const Gnuplot&);
	Gnuplot& operator=(const Gnuplot&);

	// send any command to gnuplot
	void operator ()(const std::string& command);

	void reset() { operator()("reset"); }
	void replot() { operator()("replot"); }
	void set_title(const std::string& title);

	void plot(const std::string& fname, std::size_t x, std::size_t y);
	void splot(const std::string& fname, std::size_t x, std::size_t y, std::size_t z);

protected:
	FILE* gnuplotpipe;
};

#ifndef INDIVIDUAL__
#define INDIVIDUAL__

#include <vector>
#include <ostream>

// --------------------------------------------------------------------
//                      CIndividual
// --------------------------------------------------------------------

class BProblem;

class CIndividual
{
public:
	typedef double TGene;
	typedef std::vector<TGene> TDecVec;
	typedef std::vector<double> TObjVec;

	explicit CIndividual(std::size_t num_vars = 0, std::size_t num_objs = 0);

	TDecVec& vars() { return variables_; }
	const TDecVec& vars() const { return variables_; }

	TObjVec& objs() { return objectives_; }
	const TObjVec& objs() const { return objectives_; }

	TObjVec& conv_objs() { return converted_objectives_; }
	const TObjVec& conv_objs() const { return converted_objectives_; }

	// if a target problem is set, memory will be allocated accordingly in the constructor
	static void SetTargetProblem(const BProblem& p) { target_problem_ = &p; }
	static const BProblem& TargetProblem();


private:
```

```cpp
	TDecVec variables_;
	TObjVec objectives_;
	TObjVec converted_objectives_;

	static const BProblem* target_problem_;
};



std::ostream& operator << (std::ostream& os, const CIndividual& indv);

#endif

#ifndef INITIALIZATION__
#define INITIALIZATION__


// ---------------------------------------------------------------------
//			CRandomInitialization
// ---------------------------------------------------------------------

class CIndividual;
class CPopulation;
class BProblem;

class CRandomInitialization
{
public:
	void operator()(CPopulation* pop, const BProblem& prob) const;
	void operator()(CIndividual* indv, const BProblem& prob) const;
};

extern CRandomInitialization RandomInitialization;

#endif


#ifndef POPULATION__
#define POPULATION__

// #include "individual.h"

#include <vector>

class CPopulation
{
public:
	explicit CPopulation(std::size_t s = 0) :individuals_(s) {}

	CIndividual& operator[](std::size_t i) { return individuals_[i]; }
	const CIndividual& operator[](std::size_t i) const { return individuals_[i]; }

	std::size_t size() const { return individuals_.size(); }
	void resize(size_t t) { individuals_.resize(t); }
	void push_back(const CIndividual& indv) { individuals_.push_back(indv); }
	void clear() { individuals_.clear(); }

private:
	std::vector<CIndividual> individuals_;
};

#endif
```

```cpp
#ifndef ENVIRONMENTAL_SELECTION__
#define ENVIRONMENTAL_SELECTION__
#include <vector>


// ----------------------------------------------------------------------
//		The environmental selection mechanism is the key innovation of
//  the NSGA-III algorithm.
//
//  Check Algorithm I in the original paper of NSGA-III.
// ----------------------------------------------------------------------

class CPopulation;
class CReferencePoint;

void SurvivorSelection(CPopulation *pnext, // population in the next generation
					   CPopulation *pcur,  // population in the current generation
					   size_t PopSize);

#endif

#ifndef NONDOMINATED_SORT__
#define NONDOMINATED_SORT__

#include <vector>

class BComparator;
class CPopulation;

class CNondominatedSort
{
public:
	explicit CNondominatedSort(const BComparator &d):dominate(d) {}

	// prohibit copying (VS2012 does not support 'delete')
	CNondominatedSort(const CNondominatedSort &);
	CNondominatedSort & operator= (const CNondominatedSort &);

	typedef std::vector<std::size_t> TFrontMembers; // a set of indices of individuals in a certain front
	typedef std::vector<TFrontMembers> TFronts; // a set of fronts

	std::pair< std::vector<size_t>,TFronts> operator()(const CPopulation &pop) const;

private:
	const BComparator &dominate;
};

extern CNondominatedSort NondominatedSort;

#endif

#ifndef LOG__
#define LOG__

#include <string>
#include <fstream>  // Include <fstream> for std::ios_base and std::ios_base::openmode

class CPopulation;
class Gnuplot;

// Save a population into the designated file.
bool SaveToFile(const std::string& fname, const CPopulation& pop, std::ios_base::openmode mode = std::ios_base::app);
```

```
// Show a population by calling gnuplot.
bool ShowPopulation(Gnuplot& gplot, const CPopulation&, const std::string& legend = "pop");

#endif

#ifndef MUTATION__
#define MUTATION__

// ----------------------------------------------------------------------------------
//                      CPolynomialMutation : polynomial mutation
// ----------------------------------------------------------------------------------

class CIndividual;

class CPolynomialMutation
{
public:
	explicit CPolynomialMutation(double mr = 0.0, double eta = 20) :mr_(mr), eta_(eta) {}

	void SetMutationRate(double mr) { mr_ = mr; }
	double MutationRate() const { return mr_; }
	void SetDistributionIndex(double eta) { eta_ = eta; }
	double DistributionIndex() const { return eta_; }

	bool operator()(CIndividual* c, double mr, double eta) const;
	bool operator()(CIndividual* c) const
	{
		return operator()(c, mr_, eta_);
	}

private:
	double mr_, // mutation rate
		eta_; // distribution index
};

#endif

#ifndef CROWDING_DISTANCE_H
#define CROWDING_DISTANCE_H

// #include "nondominated_sort.h"
// #include "population.h"
#include <vector>

std:: vector<double> CalculateCrowdingDistance(CPopulation& cur, CNondominatedSort::TFronts& fronts);
void SortBasedOnCrowdingDistance(CPopulation& cur, CNondominatedSort::TFronts& fronts);

#endif

#ifndef NSGAII__
#define NSGAII__

#include <cstddef>
#include <string>

// ----------------------------------------------------------------------------------
//          NSGAII
// Taken from NSGA III
// Deb and Jain, "An Evolutionary Many-Objective Optimization Algorithm Using
// Reference-point Based Non-dominated Sorting Approach, Part I: Solving Problems with
// Box Constraints," IEEE Transactions on Evolutionary Computation, to appear.
//
// http://dx.doi.org/10.1109/TEVC.2013.2281535
```

```
// ---------------------------------------------------------------------------------

class BProblem;
class CPopulation;

class CNSGAII
{
public:
	explicit CNSGAII(const std::string& inifile_name = "");
	void Solve(CPopulation* solutions, const BProblem& prob);

	const std::string& name() const { return name_; }
private:
	std::string name_;
	std::size_t obj_division_p_;
	std::size_t gen_num_;
	double  pc_, // crossover rate
		pm_, // mutation rate
		eta_c_, // eta in SBX
		eta_m_; // eta in Polynomial Mutation
};


#endif

// end of headers

// #include "nsgaii.h"
// #include "base_problem.h"
// #include "individual.h"
// #include "population.h"

// #include "initialization.h"
// #include "crossover.h"
// #include "mutation.h"
// #include "survivor_selection.h"

// #include "gnuplot_interface.h"
// #include "log.h"
#include "windows.h" // for Sleep()

#include <vector>
#include <fstream>
#include<iostream>

using namespace std;

CNSGAII::CNSGAII(const string& inifile_name) :
	name_("NSGAII"),
	obj_division_p_(12),
	gen_num_(1000),
	pc_(1.0), // default setting in NSGA-II
	eta_c_(30), // default setting
	eta_m_(20) // default setting
{
	if (inifile_name == "") return;

	ifstream inifile(inifile_name);
	if (!inifile) return;

	string dummy;
	inifile >> dummy >> dummy >> name_;
	inifile >> dummy >> dummy >> obj_division_p_;
```

```
	inifile >> dummy >> dummy >> gen_num_;
	inifile >> dummy >> dummy >> pc_;
	inifile >> dummy >> dummy >> eta_c_;
	inifile >> dummy >> dummy >> eta_m_;
}
// ----------------------------------------------------------------------
void CNSGAII::Solve(CPopulation* solutions, const BProblem& problem)
{
	CIndividual::SetTargetProblem(problem);


	size_t PopSize = 50;
	while (PopSize % 4) PopSize += 1;

	//CPopulation pop[2] = { CPopulation(PopSize) };
	std::vector<CPopulation> pop(2, CPopulation(PopSize));
	CSimulatedBinaryCrossover SBX(pc_, eta_c_);
	CPolynomialMutation PolyMut(1.0 / problem.num_variables(), eta_m_);

	Gnuplot gplot;

	int cur = 0, next = 1;

	//std::cout << problem.lower_bounds()[0] << '\n';
	RandomInitialization(&pop[cur], problem);
	for (size_t i = 0; i < PopSize; i += 1)
	{
		problem.Evaluate(&pop[cur][i]);
	}

	for (size_t t = 0; t < gen_num_; t += 1)
	{
		pop[cur].resize(PopSize * 2);

		for (size_t i = 0; i < PopSize; i += 2)
		{
			int father = rand() % PopSize,
				mother = rand() % PopSize;

			SBX(&pop[cur][PopSize + i], &pop[cur][PopSize + i + 1], pop[cur][father], pop[cur][mother]);

			PolyMut(&pop[cur][PopSize + i]);
			PolyMut(&pop[cur][PopSize + i + 1]);

			problem.Evaluate(&pop[cur][PopSize + i]);
			problem.Evaluate(&pop[cur][PopSize + i + 1]);

		}

		SurvivorSelection(&pop[next], &pop[cur], PopSize);

		//ShowPopulation(gplot, pop[next], "pop"); Sleep(10);

		std::swap(cur, next);
	}

	*solutions = pop[cur];
}

// #include "comparator.h"
// #include "individual.h"

// ---------------------------------------------------------------------------------
```

```
CParetoDominate ParetoDominate;

// ---------------------------------------------------------------------------------
//                                                         CParetoDominate
// ---------------------------------------------------------------------------------
bool CParetoDominate::operator()(const CIndividual& l, const CIndividual& r) const
{
	bool better = false;
	for (size_t f = 0; f < l.objs().size(); f += 1)
	{
		if (l.objs()[f] > r.objs()[f])
			return false;
		else if (l.objs()[f] < r.objs()[f])
			better = true;
	}

	return better;

}// CParetoDominate::operator()
// ---------------------------------------------------------------------------------




#include <vector>
#include <limits>
// #include "math_aux.h"

using namespace std;

namespace MathAux
{

// --------------------------------------------------------------------
// ASF: Achivement Scalarization Function
// --------------------------------------------------------------------
double ASF(const vector<double> &objs, const vector<double> &weight)
{
	double max_ratio = -numeric_limits<double>::max();
	for (size_t f=0; f<objs.size(); f+=1)
	{
		double w = weight[f]?weight[f]:0.00001;
		max_ratio = std::max(max_ratio, objs[f]/w);
	}
	return max_ratio;
}
// -------------------------------------------------------------------




}// namespace MathAux



// #include "individual.h"
// #include "base_problem.h"

using std::size_t;


const BProblem* CIndividual::target_problem_ = 0;
```

```
// ----------------------------------------------------------------------
CIndividual::CIndividual(std::size_t num_vars, std::size_t num_objs) :
	variables_(num_vars),
	objectives_(num_objs),
	converted_objectives_(num_objs)
{
	if (target_problem_ != 0)
	{
		variables_.resize(target_problem_->num_variables());
		objectives_.resize(target_problem_->num_objectives());
		converted_objectives_.resize(target_problem_->num_objectives());
	}
}
// ----------------------------------------------------------------------
const BProblem& CIndividual::TargetProblem() { return *target_problem_; }
// ----------------------------------------------------------------------
std::ostream& operator << (std::ostream& os, const CIndividual& indv)
{
	for (size_t i = 0; i < indv.vars().size(); i += 1)
	{
		os << indv.vars()[i] << ' ';
	}

	os << " => ";
	for (size_t f = 0; f < indv.objs().size(); f += 1)
	{
		os << indv.objs()[f] << ' ';
	}

	return os;
}

// ----------------------------------------------------------------------

// #include "initialization.h"
// #include "base_problem.h"
// #include "individual.h"
// #include "population.h"
// #include "math_aux.h"

#include <cstddef>
using std::size_t;

CRandomInitialization RandomInitialization;

void CRandomInitialization::operator()(CIndividual* indv, const BProblem& prob) const
{
	CIndividual::TDecVec& x = indv->vars();
	x.resize(prob.num_variables());
	for (size_t i = 0; i < x.size(); i += 1)
	{
		x[i] = MathAux::random(prob.lower_bounds()[i], prob.upper_bounds()[i]);
	}
}
// ----------------------------------------------------------------------
void CRandomInitialization::operator()(CPopulation* pop, const BProblem& prob) const
{
	for (size_t i = 0; i < pop->size(); i += 1)
	{
		(*this)(&(*pop)[i], prob);
	}
}
```

```cpp
// #include "crossover.h"
// #include "individual.h"
// #include "math_aux.h"
// #include "base_problem.h"

#include <cmath>
#include <algorithm>
#include <cstddef>
using std::size_t;

// ----------------------------------------------------------------------
// The implementation was adapted from the code of function realcross() in crossover.c
// http://www.iitk.ac.in/kangal/codes/nsga2/nsga2-gnuplot-v1.1.6.tar.gz
//
// ref: http://www.slideshare.net/paskorn/simulated-binary-crossover-presentation#
// ----------------------------------------------------------------------
double CSimulatedBinaryCrossover::get_betaq(double rand, double alpha, double eta) const
{
	double betaq = 0.0;
	if (rand <= (1.0 / alpha))
	{
		betaq = std::pow((rand * alpha), (1.0 / (eta + 1.0)));
	}
	else
	{
		betaq = std::pow((1.0 / (2.0 - rand * alpha)), (1.0 / (eta + 1.0)));
	}
	return betaq;
}
// ----------------------------------------------------------------------
bool CSimulatedBinaryCrossover::operator()(CIndividual* child1,
	CIndividual* child2,
	const CIndividual& parent1,
	const CIndividual& parent2,
	double cr,
	double eta) const
{
	*child1 = parent1;
	*child2 = parent2;

	if (MathAux::random(0.0, 1.0) > cr) return false; // not crossovered

	CIndividual::TDecVec& c1 = child1->vars(), & c2 = child2->vars();
	const CIndividual::TDecVec& p1 = parent1.vars(), & p2 = parent2.vars();

	for (size_t i = 0; i < c1.size(); i += 1)
	{
		if (MathAux::random(0.0, 1.0) > 0.5) continue; // these two variables are not crossovered
		if (std::fabs(p1[i] - p2[i]) <= MathAux::EPS) continue; // two values are the same

		double y1 = std::min(p1[i], p2[i]),
			y2 = std::max(p1[i], p2[i]);

		double lb = CIndividual::TargetProblem().lower_bounds()[i],
			ub = CIndividual::TargetProblem().upper_bounds()[i];

		double rand = MathAux::random(0.0, 1.0);
```

```
            // child 1
            double beta = 1.0 + (2.0 * (y1 - lb) / (y2 - y1)),
                    alpha = 2.0 - std::pow(beta, -(eta + 1.0));
            double betaq = get_betaq(rand, alpha, eta);

            c1[i] = 0.5 * ((y1 + y2) - betaq * (y2 - y1));

            // child 2
            beta = 1.0 + (2.0 * (ub - y2) / (y2 - y1));
            alpha = 2.0 - std::pow(beta, -(eta + 1.0));
            betaq = get_betaq(rand, alpha, eta);

            c2[i] = 0.5 * ((y1 + y2) + betaq * (y2 - y1));

            // boundary checking
            c1[i] = std::min(ub, std::max(lb, c1[i]));
            c2[i] = std::min(ub, std::max(lb, c2[i]));

            if (MathAux::random(0.0, 1.0) <= 0.5)
            {
                    std::swap(c1[i], c2[i]);
            }
        }

        return true;
}// CSimulatedBinaryCrossover


// #include "mutation.h"
// #include "individual.h"
// #include "math_aux.h"
// #include "base_problem.h"

#include <cstddef>
#include <algorithm>
using std::size_t;

// ----------------------------------------------------------------------
// The implementation was adapted from the code of function real_mutate_ind() in mutation.c in
// http://www.iitk.ac.in/kangal/codes/nsga2/nsga2-gnuplot-v1.1.6.tar.gz
//
// ref: http://www.slideshare.net/paskorn/simulated-binary-crossover-presentation#
// ----------------------------------------------------------------------
bool CPolynomialMutation::operator()(CIndividual* indv, double mr, double eta) const
{
    //int j;
    //double rnd, delta1, delta2, mut_pow, deltaq;
    //double y, yl, yu, val, xy;

    bool mutated = false;

    CIndividual::TDecVec& x = indv->vars();

    for (size_t i = 0; i < x.size(); i += 1)
    {
        if (MathAux::random(0.0, 1.0) <= mr)
        {
            mutated = true;

            double y = x[i],
                lb = CIndividual::TargetProblem().lower_bounds()[i],
                ub = CIndividual::TargetProblem().upper_bounds()[i];
```

```
        double delta1 = (y - lb) / (ub - lb),
            delta2 = (ub - y) / (ub - lb);

        double mut_pow = 1.0 / (eta + 1.0);

        double rnd = MathAux::random(0.0, 1.0), deltaq = 0.0;
        if (rnd <= 0.5)
        {
            double xy = 1.0 - delta1;
            double val = 2.0 * rnd + (1.0 - 2.0 * rnd) * (std::pow(xy, (eta + 1.0)));
            deltaq = std::pow(val, mut_pow) - 1.0;
        }
        else
        {
            double xy = 1.0 - delta2;
            double val = 2.0 * (1.0 - rnd) + 2.0 * (rnd - 0.5) * (std::pow(xy, (eta + 1.0)));
            deltaq = 1.0 - (std::pow(val, mut_pow));
        }

        y = y + deltaq * (ub - lb);
        y = std::min(ub, std::max(lb, y));

        x[i] = y;
      }
   }

   return mutated;
}// CPolynomialMutation


// #include "comparator.h"
// #include "nondominated_sort.h"
// #include "population.h"

using namespace std;

CNondominatedSort NondominatedSort(ParetoDominate);
// ----------------------------------------------------------------------

std::pair< std::vector<size_t>,std::vector< CNondominatedSort::TFrontMembers > > CNondominatedSort::operator()(const
CPopulation &pop) const
{
			CNondominatedSort::TFronts fronts;
			size_t num_assigned_individuals = 0;
			size_t rank = 1;
			vector<size_t> indv_ranks(pop.size(), 0);

			while (num_assigned_individuals < pop.size())
			{
				CNondominatedSort::TFrontMembers cur_front;

				for (size_t i=0; i<pop.size(); i+=1)
				{
					if (indv_ranks[i] > 0) continue; // already assigned a rank

					bool be_dominated = false;
					for (size_t j=0; j<cur_front.size(); j+=1)
					{
						if ( dominate(pop[ cur_front[j] ], pop[i]) ) // i is dominated
						{
							be_dominated = true;
							break;
```

```
					}
					else if ( dominate(pop[i], pop[ cur_front[j] ]) ) // i dominates a member in the current front
					{
						cur_front.erase(cur_front.begin()+j);
						j -= 1;
					}
				}
				if (!be_dominated)
				{
					cur_front.push_back(i);
				}
			}

			for (size_t i=0; i<cur_front.size(); i+=1)
			{
				indv_ranks[ cur_front[i] ] = rank;
			}
			fronts.push_back(cur_front);
			num_assigned_individuals += cur_front.size();

			rank += 1;
		}
		return { indv_ranks,fronts };

}// CNondominatedSort::operator()
// ----------------------------------------------------------------------




// #include "crowding_distance.h"
#include <iostream>
#include <limits>
#include <algorithm>

std::vector<double> CalculateCrowdingDistance(CPopulation& cur, CNondominatedSort::TFronts& fronts)
{
    const size_t num_objs = cur[0].objs().size();
    const size_t num_individuals = cur.size();
    const size_t num_fronts = fronts.size();

    std::vector<double> crowding_distance(num_individuals, 0.0);

    for (size_t f = 0; f < num_objs; ++f)
    {
        for (size_t i = 0; i < num_fronts; ++i)
        {
            const size_t front_size = fronts[i].size();
            if (front_size <= 2)
            {
                // Skip fronts with 2 or fewer individuals as their crowding distance will be infinity
					crowding_distance[fronts[i][0]]=std::numeric_limits<double>::infinity();
					if(front_size==2)crowding_distance[fronts[i][1]]=std::numeric_limits<double>::infinity();
                continue;
            }

            std::sort(fronts[i].begin(), fronts[i].end(), [&](size_t a, size_t b) {
                return cur[a].objs()[f] < cur[b].objs()[f];
                });

            crowding_distance[fronts[i][0]] = crowding_distance[fronts[i][front_size - 1]] = std::numeric_limits<double>::infinity();
            const double obj_range = cur[fronts[i][front_size - 1]].objs()[f] - cur[fronts[i][0]].objs()[f];
```

```
            for (size_t j = 1; j < front_size - 1; ++j)
            {
                const size_t indv_index = fronts[i][j];
                crowding_distance[indv_index] += (cur[fronts[i][j + 1]].objs()[f] - cur[fronts[i][j - 1]].objs()[f]) / obj_range;
            }
        }
    }
    return crowding_distance;
}

void SortBasedOnCrowdingDistance(CPopulation& cur, CNondominatedSort::TFronts& fronts)
{
    std::vector<double> crowding_distance = CalculateCrowdingDistance(cur, fronts);

    // Sort individuals within each front based on crowding distance
    for (size_t i = 0; i < fronts.size(); ++i)
    {
        CNondominatedSort::TFrontMembers& front = fronts[i];
        std::sort(front.begin(), front.end(), [&](size_t a, size_t b) {
         return crowding_distance[a] > crowding_distance[b]; // Sort in descending order of crowding distance
         });
    }

}



// #include "survivor_selection.h"
// #include "population.h"
// #include "math_aux.h"
// #include "nondominated_sort.h"
// #include "crowding_distance.h"

#include <limits>
#include <algorithm>

using namespace std;

// ---------------------------------------------------------------------
// SurvivorSelection():
// ---------------------------------------------------------------------
void SurvivorSelection(CPopulation *pnext, CPopulation *pcur,  size_t PopSize)
{
        CPopulation &cur = *pcur, &next = *pnext;
        next.clear();

        // ---------- Step 4 in Algorithm 1: non-dominated sorting ----------
        CNondominatedSort::TFronts fronts = NondominatedSort(cur).second;

        SortBasedOnCrowdingDistance(cur, fronts);


        for (size_t t = 0; t < fronts.size() - 1; t += 1)
        {
                for (size_t i = 0; i < fronts[t].size(); i += 1) {
                  if (next.size() >= PopSize) break;
                  next.push_back(cur[fronts[t][i]]);
                }
        }

}
// ---------------------------------------------------------------------
```

```cpp
// #include "gnuplot_interface.h"
#include <iostream>
#include <sstream>

using namespace std;

// --------------------------------------------------------
// Ref:
// http://user.frdm.info/ckhung/b/ma/gnuplot.php
// --------------------------------------------------------

Gnuplot::Gnuplot()
{
	// with -persist option you will see the windows as your program ends
	//gnuplotpipe=_popen("gnuplot -persist","w");
	//without that option you will not see the window

	// because I choose the terminal to output files so I don't want to see the window

	gnuplotpipe = _popen("gnuplot", "w");

	if (!gnuplotpipe)
	{
		cerr << ("Gnuplot not found !");
	}
}
// --------------------------------------------------------
Gnuplot::~Gnuplot()
{
	fprintf(gnuplotpipe, "exit\n");
	_pclose(gnuplotpipe);
}
// --------------------------------------------------------
void Gnuplot::operator()(const string& command)
{
	fprintf(gnuplotpipe, "%s\n", command.c_str());
	fflush(gnuplotpipe);
	// flush is necessary, nothing gets plotted else
}
// --------------------------------------------------------
void Gnuplot::set_title(const std::string& title)
{
	ostringstream ss;
	ss << "set title \"" << title << "\"";
	operator()(ss.str());
}
// --------------------------------------------------------
void Gnuplot::plot(const std::string& fname, std::size_t x, std::size_t y)
{
	ostringstream ss;

	ss << "plot \"" << fname << "\" using " << x << ":" << y;
	operator()(ss.str());
}
// --------------------------------------------------------
void Gnuplot::splot(const std::string& fname, std::size_t x, std::size_t y, std::size_t z)
{
	ostringstream ss;

	ss << "splot \"" << fname << "\" using " << x << ":" << y << ":" << z;
	operator()(ss.str());
```

```
}
// -----------------------------------------------------



// #include "assignment_problem.h"
// #include "individual.h"
// #include "math_aux.h"

#include <cmath>
#include <vector>
#include<iostream>

using std::size_t;
using std::cos;

// ----------------------------------------------------------------
//                    CProblemAssignment
// ----------------------------------------------------------------

CProblemAssignment::CProblemAssignment(size_t M, size_t k, const std::string& name, double lbs, double ubs) :
	BProblem(name),
	M_(M),
	k_(k)
{
	lbs_.resize( k_, lbs); // lower bound
	ubs_.resize( k_, ubs); // upper bound
}


bool CProblemAssignment5::Evaluate(CIndividual* indv) const
{
	CIndividual::TDecVec& x = indv->vars();
	CIndividual::TObjVec& f = indv->objs();

	if (x.size() != k_) return false; // #variables does not match

	f.resize(M_, 0);
	for (size_t i = 1; i < k_; ++i) {
	f[0] += -10 * exp(-0.2 * sqrt(MathAux::square(x[i - 1]) + MathAux::square(x[i])));
	}
	for (size_t i = 0; i < k_; ++i) {
	f[1] += pow(abs(x[i]), 0.8) + 5 * sin(pow(x[i], 3));
	}


	return true;
}


bool CProblemAssignment6::Evaluate(CIndividual* indv) const
{
	CIndividual::TDecVec& x = indv->vars();
	CIndividual::TObjVec& f = indv->objs();
	//std::cout << (x.size() == k_)<<'\n';
	if (x.size() != k_) return false; // #variables does not match

	f.resize(M_, 0);
	f[0] = MathAux::square(x[0]);
	f[1] = MathAux::square(x[0] - 2);

	return true;
}
```

```cpp
// #include "log.h"
// #include "population.h"
// #include "gnuplot_interface.h"

#include <fstream>
#include<iostream>
using namespace std;

#define OUTPUT_DECISION_VECTOR

bool SaveToFile(const std::string& fname, const CPopulation& pop, ios_base::openmode mode)
{
	ofstream ofile(fname.c_str(), mode);
	if (!ofile) return false;

	for (size_t i = 0; i < pop.size(); i += 1)
	{
#ifdef OUTPUT_DECISION_VECTOR
		for (size_t j = 0; j < pop[i].vars().size(); j += 1)
		{
			ofile << pop[i].vars()[j] << ' ';
		}
#endif

		for (size_t f = 0; f < pop[i].objs().size(); f += 1)
		{
			ofile << pop[i].objs()[f] << ' ';
		}
		ofile << endl;
	}
	ofile << endl;
	return true;
}
// ---------------------------------------------------------------------
bool ShowPopulation(Gnuplot& gplot, const CPopulation& pop, const std::string& legend)
{
	if (!SaveToFile(legend, pop, ios_base::out)) return false;


	size_t n = 0;
#ifdef OUTPUT_DECISION_VECTOR
	n = pop[0].vars().size();
#endif

	if (pop[0].objs().size() == 2)
	{
		gplot.plot(legend, n + 1, n + 2);
		return true;
	}
	else if (pop[0].objs().size() == 3)
	{
		gplot.splot(legend, n + 1, n + 2, n + 3);
		return true;
	}
	else
		return false;
}

// #include "assignment_problem.h"
// #include "nsgaii.h"
// #include "population.h"
```

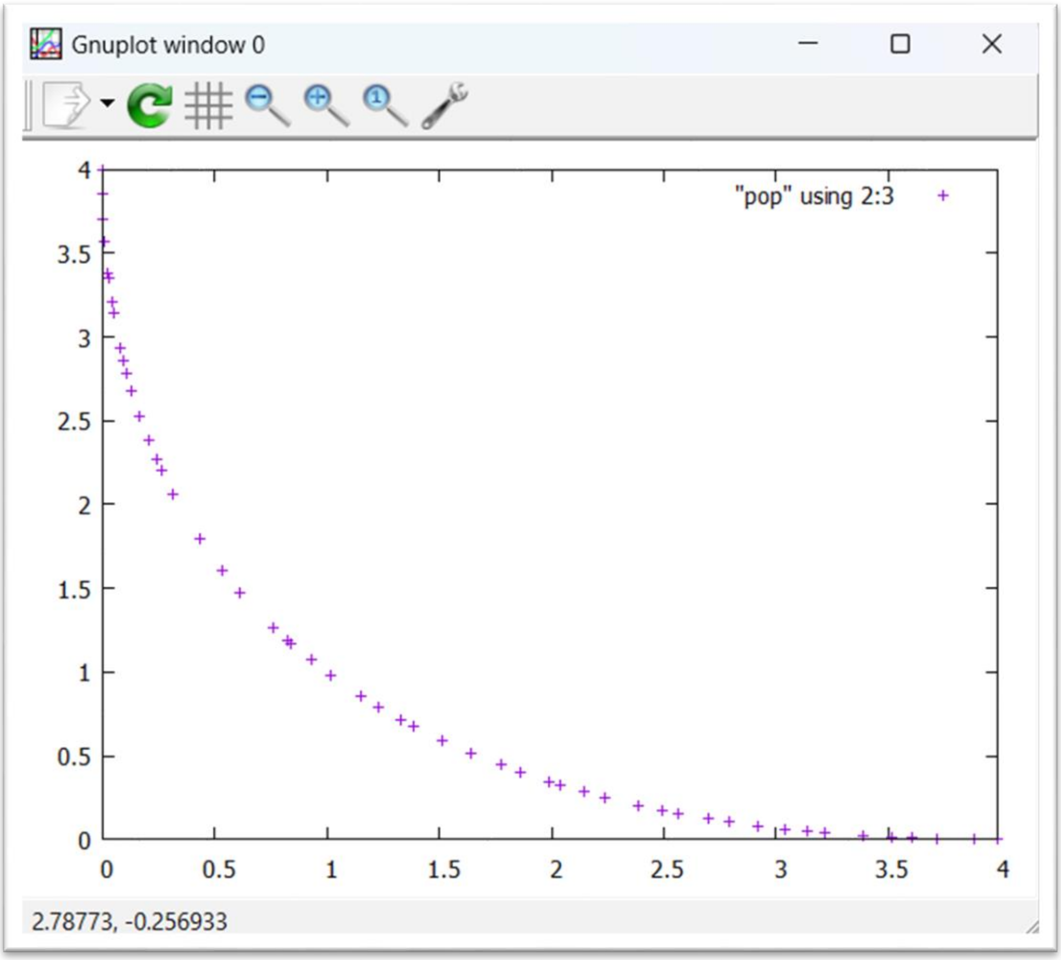
```cpp
// #include "gnuplot_interface.h"
// #include "log.h"

#include <ctime>
#include <cstdlib>
#include <iostream>

// #include "individual.h"
// #include "math_aux.h"

using namespace std;

int main()
{
	CNSGAII nsgaii("nsgaii");
	CPopulation solutions;
	Gnuplot gplot;

	const size_t NumRuns = 10;

	BProblem* problem5 = new CProblemAssignment5(2, 3,-5,5);
	BProblem* problem6 = new CProblemAssignment6(2, 1,-10,10);

	BProblem* problem = problem6;

	for (size_t r = 0; r < NumRuns; r += 1)
	{
		srand(r); cout << "Run Number: " << r << endl;

		nsgaii.Solve(&solutions, *problem);

		SaveToFile(nsgaii.name() + "-" + problem->name() + ".txt", solutions);
		ShowPopulation(gplot, solutions, "pop"); // system("pause");
	}
	delete problem;

	return 0;
}
```

Input: Terminal Input is not required Everything is Automated. If you want to change the parameters please update the main function and nsgaii function.

You can also add nsgaii.ini file and write parameter values.

Output:

Values of F1 and F2 after Max iteration



The Values after the max iteration (x, f1, f2)

```
5.94563e-05 3.53505e-09 3.99976
1.37156 1.88119 0.394933
1.27559 1.62714 0.524763
1.8067 3.26415 0.0373662
1.73708 3.01746 0.0691253
0.603055 0.363676 1.95145
1.65207 2.72932 0.121059
0.736311 0.542154 1.59691
1.50495 2.26486 0.245078
1.57025 2.46569 0.184684
1.57936 2.49437 0.176939
1.90792 3.64016 0.00847854
0.312491 0.0976508 2.84769
1.84434 3.4016 0.0242289
1.44951 2.10108 0.303038
1.70106 2.89361 0.089364
1.17717 1.38572 0.677055
1.87441 3.51343 0.015772
0.644852 0.415834 1.83643
0.547617 0.299884 2.10942
0.832125 0.692432 1.36393
0.222353 0.0494411 3.16003
0.486889 0.237061 2.28951
0.277674 0.0771029 2.96641
```

Hands On calculations:

Number of populations =4
Tournament parent selection
SBX crossover
Polynomial Mutation
Survivor selection Based on Crowding Distance and Rank

Generation Number =1

Parent selection

Sr. No.	x1	f1(x)	f2(x)
1	-9.97681	99.5367	143.444
2	-5.28855	27.9688	53.123
3	2.96304	8.77962	0.92745
4	-8.51253	72.4631	110.513

Crossover

Father	Mother		c1		c2
			x1		x1
4	2	1	-8.51253	2	-5.28855
3	2	3	-5.20879	4	2.88328

Mutation

	Mutated Value		
Sr. No.	x1	f1(x)	f2(x)
1	-6.67101	44.5024	75.1864
2	-4.12765	17.0375	37.5481
3	-5.31273	28.2251	53.476
4	5.4853	30.0885	12.1473

Survivor Selection

Sr. No.	x1	f1(x)	f2(x)	Rank	Crowding Distance	Next Generation
1	-9.97681	8.77962	0.92745	7	inf	3
2	-5.28855	17.0375	37.5481	3	inf	6
3	2.96304	30.0885	12.1473	1	inf	8
4	-8.51253	27.9688	53.123	6	inf	2
5	-6.67101	75.0785	44.4194	5	inf	
6	-4.12765	70.0454	107.523	2	inf	
7	-5.31273	7.37553	0.51236	4	inf	
8	5.4853	61.6146	97.0125	2	inf	

Generation Number =2

Parent selection

Sr. No.	x1	f1(x)	f2(x)
1	2.96304	8.77962	0.92745
2	-4.12765	17.0375	37.5481
3	5.4853	30.0885	12.1473
4	-5.28855	27.9688	53.123

Crossover

Father	Mother		c1		c2
			x1		x1
4	3	1	5.74123	2	-5.54449
1	4	3	2.96304	4	-5.28855

Mutation

	Mutated Value		
Sr. No.	x1	f1(x)	f2(x)
1	8.66478	75.0785	44.4194
2	-8.36931	70.0454	107.523
3	2.71579	7.37553	0.51236
4	-7.84949	61.6146	97.0125

Survivor Selection

Sr. No.	x1	f1(x)	f2(x)	Rank	Crowding Distance	Next Generation
1	-9.97681	8.77962	0.92745	2	inf	7
2	-5.28855	17.0375	37.5481	3	inf	1
3	2.96304	30.0885	12.1473	3	inf	2
4	-8.51253	27.9688	53.123	4	inf	3
5	8.66478	75.0785	44.4194	4	inf	
6	-8.36931	70.0454	107.523	6	inf	
7	2.71579	7.37553	0.51236	1	inf	
8	-7.84949	61.6146	97.0125	5	inf	

Best Answer: x = [2.71579] (Randomly selected from pareto-optimal front)


Thank You!