

Mixed-Integer Constrained Optimization: An Evolutionary Approach

M.J. Pollard

September 10, 2007

Contents

1	Introduction	1
2	Motivation	3
3	Problem Definition	5
3.1	Mathematical Programming	5
3.1.1	Linear Programming vs. Non-Linear Programming	5
3.1.2	Integer Programming vs. Real Valued Programming vs. Mixed Integer Programming	6
3.2	Mixed Integer Non-Linear Programming	6
4	Current Methods	7
4.1	Branch and Bound	7
4.2	Outer Approximation	8
5	Evolutionary Strategies	9
5.1	Evolutionary Operators	9
5.2	Outline	9
5.3	Algorithm Parameters	10
5.4	Representation	10
5.5	Operators	11
5.5.1	Recombination	11
5.5.2	Mutation	12
5.5.3	Selection	13
5.6	Termination Criteria	13
6	Results	15
6.1	Method	15
6.2	Algorithm parameters	15
6.3	Test Functions	16
6.4	Linear Programming Results	16
6.4.1	Linear Programming Problem 1	17
6.4.2	Linear Programming Problem 2	17
6.4.3	Linear Programming Problem 3	18
6.5	Mixed Integer Non-Linear Programming Results	19
6.5.1	Problem 1	19
6.5.2	Problem 2	19
6.5.3	Problem 3	20

6.5.4	Problem 4	20
6.5.5	Problem 5	21
6.5.6	Problem 6	22
6.5.7	Problem 7	22
6.5.8	Problem 8	23
6.6	Conclusions	24
7	Conclusion	25
7.1	Equality Constraints	25
7.2	Divergence	25
7.3	Conclusion	26
8	Further Work	27
8.1	Generating feasible solutions vs. Penalty function	27
8.2	Penalty function optimization	27
8.3	Coupling the discrete and continuous vector optimization	27
9	Code	29
9.1	util.h	29
9.2	rand.h	30
9.3	rand.c	30
9.4	poputils.h	33
9.5	poputils.c	34
9.6	operators.h	41
9.7	operators.c	41
9.8	memutils.h	49
9.9	memutils.c	49
9.10	algo.h	55
9.11	algo.c	55
10	Acknowledgements	61

Chapter 1

Introduction

Mixed Integer Nonlinear Programming (MINLP) refers to mathematical programming with continuous and discrete variables and nonlinearities in the objective function and constraints.

Evolutionary strategies are an optimization technique based on ideas of adaptation and evolution.

This thesis attempts to devise an algorithm to solve MINLP problems utilizing ES methods. Firstly we define the problem area in chapter 3. Then we briefly discuss some current methods in chapter 4. Evolutionary strategies as well as our algorithm are discussed in chapter 5. Results on several problems are presented in chapter 6. Finally some possible extensions for future work are discussed in chapter 8.

Chapter 2

Motivation

Mixed integer non-linear programming (MINLP) problems are an important class of problems in the industry. While many problems can be formulated as linear programming problems (i.e. transportation problem, travelling salesman problems etc.) some problem domains have a natural non-linear and integer valued variable representation.

Examples of these are chemical process optimization [3] and image analysis agents [5].

Chemical process optimization usually employs non-linear objective and constraint functions as well as integer values variables to represent the presence or absence of a unit in the process.

The SAVAGE project deals with the optimization of image analysis agent based on training data (medical IVUS images annotated by experts). Some research has been done in utilising MINLP methods to optimize the parameters for this agent.

As evolutionary strategies for mixed integer optimisation is a relatively young field, there are little or no studies done on constrained optimization utilising these strategies.

The central question of this paper is whether it is feasible, and even possible to utilize evolutionary strategies to solve this class of problems.

Chapter 3

Problem Definition

3.1 Mathematical Programming

Mathematical programming is the application of mathematical models, in particular optimisation models, to aid in reaching decisions. Mathematical programming is a subset of operations research and there exists an enormous body of knowledge surrounding these optimisation techniques.

Modelling a real world problem is done in several steps. First we identify some object variables in the problem. An objective function is constructed. These real world variables are usually constrained by real world constraints. Examples include a limited supply of a resource, limited capacity in transporting these resources etc. By their own each variable may be bounded, from above, from below or both, and in combination with each other the variables form constraint functions.

The objective function is either to be maximized or minimized constrained to the above mentioned constraints.

The general mathematical programming problem can be stated in standard form as follows:

$$\begin{array}{llll} \min & z & = & f(\mathbf{x}) \\ \text{subject to} & g(\mathbf{x}) & \leq & \mathbf{0} \\ & h(\mathbf{x}) & = & \mathbf{0} \\ & \mathbf{x} & \in & \mathbb{X} \end{array}$$

where $z \in \mathbb{R}$ is called the objective value, $f : \mathbb{X} \rightarrow \mathbb{R}$ is called the objective function. The constraint functions $g : \mathbb{X} \rightarrow \mathbb{R}$, inequality constraints, and $h : \mathbb{X} \rightarrow \mathbb{R}$, equality constraints, define valid values for elements of \mathbb{X} . If the functions $g(\mathbf{x})$ and $h(\mathbf{x})$ are satisfied, \mathbf{x} is called a feasible solution to the problem. The set \mathbb{X} is the domain of \mathbf{x} and consists of all the feasible solutions to the problem.

The question becomes: from all $\mathbf{x} \in \mathbb{X}$ choose an \mathbf{x}_0 so that $f(\mathbf{x}_0) \leq f(\mathbf{x}) \quad \forall \mathbf{x} \in \mathbb{X}$.

3.1.1 Linear Programming vs. Non-Linear Programming

In linear programming the objective function and the constraint functions are all linear in \mathbf{x} . Several large classes of problems may be modelled using linear

functions and consequently very efficient algorithms, such as the simplex algorithm, exist to solve linear programming problems. Current commercial systems can solve problems containing several thousand variables.

In non-linear programming the objective and constraint functions may be nonlinear in \mathbf{x} . In linear programming the linear constraints result in a convex feasible solution space. Several algorithms as well as the characterization of optimal feasible solutions rely on the convexity of the solution space. When the constraint functions become non-linear, we can't guarantee the convexity of the solution space and the usefulness of the linear methods diminish.

This makes non-linear programming problems more difficult to solve. Several non-linear programming algorithms rely on linearising the constraints, and therefore the solution space and then employ linear programming methods to find an (approximate) optimal feasible solution.

3.1.2 Integer Programming vs. Real Valued Programming vs. Mixed Integer Programming

A common assumption in many methods is the divisibility of the variables. It is assumed that each variable is an element of the reals. If this is not the case we refer to integer programming. If some of the variables are reals and others integer valued, we speak of mixed integer programming.

Many of the algorithms to solve (non-)linear problems rely on the continuity of the variable space. In the light of this, integer programming algorithms solves these kind of problems iteratively by relaxing the integral constraints, solving the continuous problem and then shrinking the solution space by cutting away non-optimal regions of the variable space. The branch and bound method is an example of this.

3.2 Mixed Integer Non-Linear Programming

From section 3.1 we see that any combination of linear/non-linear and integer/real/mixed integer programming are possible. This study focuses on the most complex combination of Mixed Integer Non-Linear programming.

We now state the general MINLP problem:

$$\begin{array}{llll}
 \min & z & = & f(\mathbf{x}, \mathbf{y}) \\
 \text{subject to} & g_i(\mathbf{x}, \mathbf{y}) & \geq & \mathbf{0} \quad i \in I_g \\
 & h_j(\mathbf{x}, \mathbf{y}) & = & \mathbf{0} \quad j \in I_h \\
 & \mathbf{lb}_x & \leq & \mathbf{x} \leq \mathbf{ub}_x \\
 & \mathbf{lb}_y & \leq & \mathbf{y} \leq \mathbf{ub}_y \\
 & \mathbf{x} & \in & \mathbb{R}^n \quad n \geq 0 \\
 & \mathbf{y} & \in & \mathbb{Z}^m \quad m \geq 0
 \end{array}$$

where $z \in \mathbb{R}$ is called the objective value and $f : \mathbb{R}^n \cup \mathbb{Z}^m \rightarrow \mathbb{R}$ is called the objective function. $g : \mathbb{R}^n \cup \mathbb{Z}^m \rightarrow \mathbb{R}$ and $h : \mathbb{R}^n \cup \mathbb{Z}^m \rightarrow \mathbb{R}$ are the constraint functions, where I_g represents the index set of inequality than constraints and I_h represents the index set of equality constraints. The members of the solution space are bounded from above and below by \mathbf{ub} and \mathbf{lb} , respectively. \mathbf{x} is a real valued vector in \mathbb{R}^n and \mathbf{y} is a integer valued vector in \mathbb{Z}^m .

Chapter 4

Current Methods

There are several methods currently employed to solve MINLP problems. They differ in complexity and running time as well as quality of optimal solutions.

4.1 Branch and Bound

Solving the MINLP using branch and bound requires two main steps. First, the continuous relaxation, where the integral constraint is dropped from the problem, of the problem is solved. This description for branch and bound was taken almost verbatim from [1].

If all the discrete variables take on integer values, the problem is solved. Otherwise we do a tree based search in the integer space of the problem. The branch and bound algorithm maintains a list of unsolved subproblems that are generated from the initial problem. Also, the best known solution $(\mathbf{x}^*, \mathbf{y}^*)$, called the incumbent solution, is kept. The incumbent solution provides an upper bound on the objective value.

The branch and bound algorithm follows:

1. **Initialize:** Create a list L with the MINLP as the initial subproblem. If a good integer solution is known, then initialize $(\mathbf{x}^*, \mathbf{y}^*)$ and make a note of the objective value ub . If no incumbent solution is known, set ub to $+\infty$.
2. **Select:** Select an unsolved subproblem, T , from the list L . If L is empty, stop and return optimal solution. (if there is an incumbent solution, then it is optimal. If there is no incumbent solution, then the MINLP is infeasible.)
3. **Solve:** Relax the integer constraints in T and solve the resulting nonlinear programming relaxation. Obtain a solution $(\mathbf{x}', \mathbf{y}')$ and a lower bound lb on the optimal value of the subproblem.
4. **Fathom:** If the relaxed subproblem was infeasible, then T will clearly not yield a better solution to the MINLP than the incumbent solution. Similarly, if $lb \geq ub$, then the current subproblem cannot yield a better solution to the MINLP than the incumbent solution. Remove T from list L and return to step 2.

5. **Integer Solution:** If \mathbf{y}' is integer, then a new incumbent solution has been obtained. Update $(\mathbf{x}^*, \mathbf{y}^*)$ and ub . Remove T from L and return to step 2.
6. **Branch:** At least one of the integer variables y_k takes on a fractional value in the solution to the current subproblem T . Create a new subproblem T_1 by adding the constraint

$$y_k \leq \lfloor y_k \rfloor.$$

Create a second new subproblem T_2 by adding the constraint

$$y_k \geq \lceil y_k \rceil.$$

Remove T from L , add T_1 and T_2 to L and return to step 2.

Branch and bound is guaranteed to find the optimal solution to the problem or terminate with the conclusion that the problem is infeasible.

An advantage of this method is the clear decoupling of the continuous and discrete optimizers. Any usable continuous optimizer may be used for solving the relaxed problem while the branch and bound method searches through the discrete space for the optimal solution.

A major disadvantage of this method is that the problem needs to be solved at every node in the tree search. That is conceptually 2^n problems to solve, where n is the number of discrete variables in the problem.

The actual number of problems to solve is less than this upper bound since we may cull branches from the tree where we are certain the sub tree will not yield a better solution to the problem.

4.2 Outer Approximation

The outer approximation method approximates the non-linear space utilizing linear constraints. Supporting linear hyperplanes are calculated at each iteration of the algorithm. Since we have efficient methods of solving linear programming problems, we may utilize these to solve for the MINLP problem.

These linearizations overestimates the feasible region while at the same time the optimal solution is underestimated. Since we add a lot of constraints the problem may become intractable.

This method may be terminated at any time to obtain an approximate solution to the problem. Although this is true of any method, in this case the linearization of the feasible region approaches the actual feasible region in the limit. Once the feasible region is sufficiently approximated we can be reasonably sure that the produced answer is close to the global optimum.

Chapter 5

Evolutionary Strategies

Evolutionary Strategies refer to a method of searching for an optimal solution that borrows ideas from nature's process of evolution.

We start with a set X of solutions that has been drawn randomly from the solution space. We define several operators that operate on one or more (even the entire set X) of possible solutions. These operators are applied to the set of solutions iteratively, changing these solutions stochastically by a small amount at an iteration that results in one solution possibly presenting itself as the optimal solution.

We refer to this set X as a population. The members of this population are the individual solutions. An iteration of the algorithm is referred to as a generation.

5.1 Evolutionary Operators

There are three basic operators recognized in literature.

1. **Mutation:** Small random changes are made to each member. Each member has an equal probability of being mutated.
2. **Recombination:** Local recombination is where we randomly pick two or more members of the population and combine them to form a new member. Global recombination takes the entire population into account to create a new solution. Recombination is responsible for the growth of the population.
3. **Selection:** We deterministically select members from the current population to form the next generation.

By iteratively applying these operators we probabilistically move closer and closer the optimal solution of the problem.

5.2 Outline

Since there is little available research for the solution of MINLP problems using evolutionary strategies, the proposed algorithm implements a straight forward

version of it. These experiments are mainly to determine if it is possible to solve these problems utilizing these techniques.

The algorithm outline is as follows:

```

initialise population P(0)
// The algorithm runs the discrete mutation for n, the number of discrete
// variables, generations. This value was arbitrarily chosen to keep computation
// to a reasonable level and there was no clear guideline found for choosing
// this number. Another choice may be 2^n.
for t = 1 to (n + 1)
    R(t) = recombine discrete variables(P(t))
    M(t) = mutate discrete vector(R(t))
    E(t) = evaluate(M(t))
    S(t) = select_best(E(t))
    P(t + 1) = optimize_continuous(S(t))
    t++
done
return best_member(P(t))

optimimze_continuous(S(t))
Q(t) empty
for j = 1 to mu
    C(t) = initialise population(S(t), j)
    X(t) = continuousES(C(t))
    x_j = best_member(X(t))
    Q(t) = Q(t) union {x_j}
done
return Q(t)

```

The basic idea of this algorithm is to run an ES for the discrete variables. Each candidate solution of the discrete ES is evaluated by means of the objective function after optimizing its continuous variables by an continuous ES.

5.3 Algorithm Parameters

The algorithm parameters are as follow

1. **Population Size:** The size of each generation.
2. **Offspring per Population member:** The factor of population increase during the recombination step.
3. **Selection Type:** The method used be the selection operator.
4. **Sigma Type:** The variant of continuous mutation operator to use.

5.4 Representation

Each member consists of various variables used during the algorithm. In the C implementation we used a struct containing variables and arrays. The members consisted of:

1. **Continuous Variables:** An array of doubles containing the relevant number of variables.
2. **Discrete Variables:** An array of integers containing the relevant number of variables.
3. **Step Size:** Depending on the Sigma Type parameter this could be an array of doubles or a single double value.
4. **Step Direction:** Depending on the Sigma Type parameter this was an array of doubles or not present at all.
5. **Penalty Value:** The penalty value was a single double value that indicated if the member was feasible or not. It also measured the magnitude of the violation of constraints.
6. **Objective Value:** A double variable indicating the member's fitness.
7. **Age:** Depending on the Selection Type parameter this was an indication of how many iterations the member has 'survived'.

5.5 Operators

The operators modifies the population and are responsible for directing the search for the best member, which would represent a solution to our MINLP problem.

The variation operators employed were Recombination and Mutation, both of which consisting of two subroutines: one for the continuous and one for the discrete variables.

5.5.1 Recombination

During recombination the current population is enlarged by a factor determined by the Offspring per Population Member parameter. Recombination, as the name suggests, takes two or more members of the population, referred to as parents, and creates a new member, an offspring, containing information from all parents.

Continuous Recombination

Continuous recombination takes place during the continuous ES step of the algorithm.

For each member of the offspring population we randomly select two parents. Then for each object variable we randomly select one variable from the parents and copy it to the offspring member.

For each evolution parameter we take the average value between the two parents and assign it to the offspring member.

Since the continuous optimization takes place within the optimization of the discrete variables, and the fact that we need both discrete and continuous variables in a member to assign an objective value to it, we randomly select one of the parents' discrete vector and assign it to the offspring population member.

Discrete Recombination

Discrete recombination takes place during the discrete ES optimization.

For each member of the offspring population we randomly select two parents from the current population. Then for each object variable we randomly select one parent and copy the corresponding variable to the offspring population member.

Once again we randomly select one parent to be the source of the continuous vector for the offspring population member.

5.5.2 Mutation

The mutation operation is responsible for local search. During mutation small random changes are made to the object variables.

Continuous Mutation

We investigated three standard continuous mutation strategies. A single step size, multiple step sizes and correlated mutation.

Single step size mutation involves a single mutation parameter σ per individual. First the mutation parameter is mutated. Then each continuous variable is mutated using the new mutation parameter according to the following formula:

$$\begin{aligned}\sigma' &= \sigma * \exp(\tau_0 * N(0, 1)) \\ x'_i &= x_i * \sigma' * N_i(0, 1)\end{aligned}$$

where $\tau_0 \approx 1/\sqrt{(n)}$ and $N(0, 1)$ is a normally distributed random variable with mean 0 and standard deviation 1.

Simple mutation can be visualized as generating a random member of the solution space within a hypersphere surrounding the current solution of a size determined by the σ parameter.

Multiple step size mutation is similar to single step size mutation but each continuous variable has its own mutation parameter σ . The adaptation of variables happen according to the following:

$$\begin{aligned}\sigma'_i &= \sigma_i * \exp(\tau' * N(0, 1) + \tau * N(0, 1)) \\ x'_i &= x_i * \sigma'_i * N_i(0, 1)\end{aligned}$$

The probability density function of the distribution of offspring has elliptic iso-heightlines. The axis of the ellipsoids are axis parallel and scaled by the parameter σ_i .

Correlated mutations also have a mutation parameter for every continuous variable. During correlated mutation the mutation parameters are adapted according to a covariance matrix C . Another parameter β is added. The continuous variables are mutated according to the following equations:

$$\begin{aligned}\sigma'_i &= \sigma_i * \exp(\tau' * N(0, 1) + \tau * N_i(0, 1)) \\ \alpha'_j &= \alpha_j + \beta * N_j(0, 1) \\ x'_i &= x_i \sigma'_i + N(0, C)\end{aligned}$$

Roughly speaking, multiple step size mutation can be visualized as picking a random member of the solution space within an ellipsoid whose axes are at an

angle to the coordinate axis, determined by the α parameters, surrounding the current solution. The size of the ellipsoid is determined by each σ_i parameter.

Discrete Mutation

Discrete mutation is completely different from continuous mutation. We can't make 'small' changes to the discrete vector like we can on the continuous vector. A change of just one unit in the discrete vector can lead to a large jump in the member's objective value.

The algorithm mutates the discrete vector as follows.

The discrete vector contains n variables, each with its own upper and lower bounds (if they exist). Each variable is given the chance of $1/n$ to mutate. This is a well known mutation probability in the literature as it can be shown that this gives the largest probability that only one member of the discrete vector will be mutated.

The variables that are mutated according to this probability gets replaced by a random value from the domain of possible values for this variable.

5.5.3 Selection

The selection operator takes the result of the mutation and recombination operators and selects the next generation. This selection happens deterministically (assuming that all solutions have a unique rank w.r.t. fitness). The best N members form the next generation. In case of equally ranked solutions, a random selection may take place among members of equal rank.

Selection takes place on either the children, or on the union of the parents and the children populations.

An EA which takes into account the parents as well as the children may suffer from the following occurrence. A 'good' solution emerges after some time. This solution has the property that its objective value may be close to the global optimum, but the distance to the globally optimal solution in the search space may be high and barriers with worse objective values may lay between both solutions. It will survive for many generations due to the selection deterministically picking the best n solutions. This prevents the algorithm to explore other parts of the solution space.

As a middle route between selection with memory and without memory [3] introduce an "age" parameter into the selection. Decide on a maximum number of generations a member can survive in the population. Selection with memory is applied on the union of the parent and child populations from which the members that have reached their maximum age have been removed.

5.6 Termination Criteria

There does not exist a characterization of the optimal solution in MINLP problems that utilizes local information such as the KKT criteria for LP problems. This makes it difficult in probabilistic algorithms (such as EA) to decide when to terminate the algorithm.

In our algorithm we terminate the algorithm if two successive iterations of the algorithm are within a prescribed error margin. This indicates that we are

battling to find better solutions, and that successive generations are not making much progress.

When the best members of two successive generations are within this margin, we stop the algorithm and return the best member found during the course of the optimization run.

Chapter 6

Results

This section shows our algorithm and results.

6.1 Method

We evaluated several problems from different mathematical programming backgrounds. First we applied the algorithm to several basic artificial test problems to see if it steers us in the right direction. Then we tried a few linear programming problems that were originally motivated by real-world optimization tasks. Finally several minlp problems from [2] were tried to test applicability to these problems.

The algorithms was applied ten times to each problem.

The questions we hope to answer in this section are the following:

1. Does the algorithm actually work? Does it successfully find the global optimum of test functions and simple linear programming problems?
2. Does the algorithm find the global optimum of mixed integer non linear programming problems we chose to test it against?

The test functions were optimized first. Each test function were optimized ten times for each of the three mutation strategies mentioned in section 5.5.2. The result of these optimizations were used to choose the mutation strategy for the optimization of the remaining problems.

6.2 Algorithm parameters

The algorithm provides for several parameters to be set. These are listed below:

1. **Population size:** The size of the population to create for the evolutionary strategy. Each experiment used a population size of 15.
2. **Offspring per population member:** The number of offspring per population member to produce during the evolutionary strategy. This parameter was set to 7 to produce 105 new members each iteration.

3. **Selection Type:** The type of selection to perform as given in Chapter 5.5.3. All the experiments selected the best 15 members using the age parameter as discussed.
4. **Age Parameter:** The age parameter as discussed in 5.5.3 was set to 10 generations.
5. **Sigma Type:** The type of mutation to perform as given in Chapter 5. The test functions were optimized ten times with each of the strategies given in section 5.5.2. After considering the results correlated mutation was chosen for the remainder of the problems.

6.3 Test Functions

There exists several test function for evolutionary strategies that are popular in literature. We chose the following three to test our ES algorithm. They are presented with the results obtained:

$$z = \sum_{i=1}^n x_i^2 \quad (6.1)$$

$$z = \sum_{i=1}^n x_i^2 * i \quad (6.2)$$

$$z = \sum_{i=1}^n \sum_{j=1}^i x_i^2 \quad (6.3)$$

30 Variables		Single	Multiple	Correlated
Test1 6.1	Iterations	654	1538	627
	Evaluations	68807	161654	65961
Test2 6.2	Iterations	6672	1769	783
	Evaluations	700613	185808	82352
Test3 6.3	Iterations	6694	1437	595
	Evaluations	702954	150959	62591

Each one of these runs minimized the problem at hand to $z = 0$, which is the correct minimum value for each of these problems.

6.4 Linear Programming Results

This section lists results obtained applying the method to linear programming problems. Three problems were selected from [4]

6.4.1 Linear Programming Problem 1

The following problem appears in the book on Page 142 Example 4.3.

Minimize

$$z = x_1 - 2x_2$$

Subject to

$$\begin{aligned} x_1 + x_2 &\geq 2 \\ -x_1 + x_2 &\geq 1 \\ x_2 &\leq 3 \\ x_1, x_2 &\geq 0 \end{aligned}$$

The results of ten runs of the algorithm are summarized in the next table.

Run	Function Evaluation	Constraint Evaluations	Iterations	Objective Value
0	15225	30450	144	-6.000000
1	6195	12390	58	-4.388759
2	13650	27300	129	-6.000000
3	12810	25620	121	-6.000000
4	15645	31290	148	-6.000000
5	5775	11550	54	-3.474183
7	14175	28350	134	-6.000000
7	5670	11340	53	-4.909286
8	13230	26460	125	-6.000000
9	14070	28140	133	-6.000000
Average	11644.5	23289	109.9	

The minimum value of the objective function is $z = -6$. The algorithm found the correct value in seven out of ten runs in an average of 109.9 iterations.

6.4.2 Linear Programming Problem 2

The following problem appears in the book on Page 191 Example 5.1.

Minimize

$$z = -x_1 - 2x_2 + x_3 - x_4 - 4x_5 + 2x_6$$

Subject to

$$\begin{aligned} x_1 + x_2 + x_3 + x_4 + x_5 + x_6 &\leq 6 \\ 2x_1 - x_2 - 2x_3 + x_4 &\leq 4 \\ x_3 + x_4 + 2x_5 + x_6 &\leq 4 \\ x_1, x_2, x_3, x_4, x_5, x_6 &\geq 0 \end{aligned}$$

The results of ten runs of the algorithm are summarized in the next table.

Run	Function Evaluation	Constraint Evaluations	Iterations	Objective Value
0	84105	252315	800	-15.999980
1	73080	219240	695	-15.861890
2	75705	227115	720	-15.878410
3	78015	234045	742	-15.999050
4	74970	224910	713	-15.909280
5	78435	235305	746	-15.999880
6	84525	253575	804	-15.999960
7	87045	261135	828	-15.998230
8	93030	279090	885	-15.928490
9	85260	255780	811	-15.441860
Average	81417	244251	774.4	

The minimum value of the objective function is $z = -16$. The algorithm came very close (< 0.001) to this value four out of ten times without actually reaching it.

6.4.3 Linear Programming Problem 3

The following problem appears in the book on Page 344 Example 7.2.

Minimize

$$z = -2x_1 - x_2 - 3x_3 - x_4$$

Subject to

$$x_1 + x_2 + x_3 + x_4 \leq 6$$

$$x_1 + x_2 + x_3 + x_4 \leq 6$$

$$x_2 + 2x_3 + x_4 \leq 4$$

$$x_1 + x_2 \leq 6$$

$$x_2 \leq 2$$

$$-x_3 + x_4 \leq 3$$

$$x_3 + x_4 \leq 5$$

$$x_1, x_2, x_3, x_4 \geq 0$$

The results of ten runs of the algorithm are summarized in the next table.

Run	Function Evaluation	Constraint Evaluations	Iterations	Objective Value
0	42630	213150	405	-5.610968
1	56700	283500	539	-5.908344
2	65520	327600	623	-5.268424
3	65730	328650	625	-5.968157
4	41055	205275	390	-5.98899
5	51870	259350	493	-5.702892
6	52080	260400	495	-5.635172
7	58485	292425	556	-5.836824
8	46620	233100	443	-5.988268
9	42210	211050	401	-5.809434
Average	52290	261450	497	

The minimum value of the objective function is $z = -14$. The algorithm did not come close to this value.

6.5 Mixed Integer Non-Linear Programming Results

The problems appearing in this section comes from [2]. The M-Simpsa algorithm was able to solve all these problems.

6.5.1 Problem 1

Minimize

$$z = 2x + y$$

Subject to

$$\begin{aligned} 1.25 - x^2 - y &\leq 0 \\ x + y &\leq 1.6 \\ 0 &\leq x \leq 1.6 \\ y &\in \{0, 1\} \end{aligned}$$

The results of ten runs of the algorithm are summarized in the next table.

Run	Function Evaluation	Constraint Evaluations	Iterations	Objective Value
0	82845	165690	757	2.236068
1	82530	165060	754	2.236068
2	81480	162960	744	2.236068
3	81375	162750	743	2
4	81060	162120	740	2
5	84210	168420	770	2.236068
6	82320	164640	752	2.236068
7	81585	163170	745	2
8	82635	165270	755	2.236068
9	80850	161700	738	2
Average	82089	164178	749.8	

The minimum value of the objective function is $z = 2$. This value was reached 4 out of the 10 runs.

6.5.2 Problem 2

Minimize

$$z = -y + 2x_1 + x_2$$

Subject to

$$\begin{aligned} x_1 - 2e^{-x_2} &= 0 \\ -x_1 + x_2 + y &\leq 0 \\ 0.5 &\leq x_1 \leq 1.4 \\ y &\in \{0, 1\} \end{aligned}$$

The results of ten runs of the algorithm are summarized in the next table.

Run	Function Evaluation	Constraint Evaluations	Iterations	Objective Value
0	246120	246120	2312	2.141443
1	266805	266805	2509	2.126845
2	274995	274995	2587	2.694013
3	249480	249480	2344	2.142112
4	245070	245070	2302	2.694016
5	270690	270690	2546	2.125915
6	239505	239505	2249	2.821467
7	309120	309120	2912	2.690942
8	249690	249690	2346	2.695682
9	274050	274050	2578	2.152587
Average	262552.5	262552.5	2468.5	

The minimum value of the objective function is $z = 2.124$. The algorithm got close a few times without reaching the optimal value.

6.5.3 Problem 3

Minimize

$$z = -0.7y + 5(x_1 - 0.5)^2 + 0.8$$

Subject to

$$\begin{aligned} -exp^{x_1-0.2} - x_2 &\leq 0 \\ x_2 + 1.1y &\leq -1 \\ x_1 - 1.2y &\leq 0.2 \\ 0.2 \leq x_1 &\leq 1.0 \\ -2.22554 \leq x_2 &\leq -1.0 \\ y &= \{0, 1\} \end{aligned}$$

The results of ten runs of the algorithm are summarized in the next table.

Run	Function Evaluation	Constraint Evaluations	Iterations	Objective Value
0	152775	458325	1423	1.25
1	217560	652680	2040	1.076543
2	216825	650475	2033	1.076543
3	202020	606060	1892	1.076543
4	207270	621810	1942	1.076543
5	217035	651105	2035	1.076543
6	195720	587160	1832	1.076543
7	202650	607950	1898	1.076543
8	212520	637560	1992	1.076543
9	201495	604485	1887	1.076543
Average	202587	607761	1897.4	

The minimum value of the objective function is $z = 1.07654$. The optimal solution was reached 9 out of 10 times.

6.5.4 Problem 4

Minimize

$$z = 2.0x_1 + 3.0x_2 + 1.5y_1 + 2.0y_2 - 0.5y_3$$

Subject to

$$\begin{aligned}
 x_1^2 + y_1 &= 1.25 \\
 x_2^{1.5} + 1.5y_2 &= 3.0 \\
 x_1 + y_1 &\leq 1.6 \\
 1.333x_2 + y_2 &\leq 3.0 \\
 -y_1 - y_2 + y_3 &\leq 0 \\
 x_1, x_2 &\geq 0 \\
 y_1, y_2, y_3 &= \{0, 1\}^3
 \end{aligned}$$

The results of ten runs of the algorithm are summarized in the next table.

Run	Function Evaluation	Constraint Evaluations	Iterations	Objective Value
0	712695	2138085	6622	7.66718
1	713995	2141985	6632	7.66718
2	709679	2129037	6592	7.66718
3	703315	2109945	6534	7.66718
4	704086	2112258	6539	7.66718
5	706124	2118372	6560	7.66718
6	710858	2132574	6603	7.66718
7	711328	2133984	6610	7.66718
8	708139	2124417	6578	7.66718
9	696377	2089131	6471	7.931112
Average	707659.6	2122978.8	6574.1	

The minimum value of the objective function is $z = 7.66718$. The optimal solution was reached 9 out of 10 times.

6.5.5 Problem 5

Minimize

$$z = 7.5y_1 + 5.5y_2 + 7.0v_1 + 6.0v_2 + 5.0x$$

Subject to

$$\begin{aligned}
 y_1 + y_2 &= 1 \\
 z_1 &= 0.9(1 - e^{(-0.5v_1)})x_1 \\
 z_2 &= 0.8(1 - e^{(-0.4v_2)})x_2 \\
 x_1 + x_2 - x &= 0 \\
 z_1 + z_2 &= 10 \\
 v_1 &\leq 10y_1 \\
 v_2 &\leq 10y_2 \\
 x_1 &\leq 20y_1 \\
 x_2 &\leq 20y_2 \\
 x_1, x_2, z_1, z_2, v_1, v_2 &\geq 0 \\
 y &= \{0, 1\}^2
 \end{aligned}$$

The results of ten runs of the algorithm are summarized in the next table.

Run	Function Evaluation	Constraint Evaluations	Iterations	Objective Value
0	4488645	17954580	42701	103.3193
1	4005015	16020060	38095	100.2022
2	4077675	16310700	38787	107.5128
3	4570965	18283860	43485	117.2978
4	4033260	16133040	38364	105.0215
5	4464180	17856720	42468	108.4293
6	4632915	18531660	44075	99.29361
7	5185005	20740020	49333	99.38864
8	4547865	18191460	43265	107.637
9	4700115	18800460	44715	110.0159
Average	4470564	17882256	42528.8	

The minimum value of the objective function is $z = 99.2396$. The value was never reached, but the algorithm came close.

6.5.6 Problem 6

Minimize

$$z = (y_1 + 2.0y_2 + 3.0y_3 - y_4) * (2.0y_1 + 5.0y_2 + 3.0y_3 - 6.0y_4)$$

Subject to

$$y_1 + 2.0y_2 + y_3 + 3.0y_4 \geq 4$$

$$y = \{0, 1\}^4$$

The results of ten runs of the algorithm are summarized in the next table.

Run	Function Evaluation	Constraint Evaluations	Iterations	Objective Value
0	8400	8400	1	-6
1	8400	8400	1	-6
2	8400	8400	1	-6
3	8400	8400	1	-6
4	8400	8400	1	-6
5	8400	8400	1	-6
6	8400	8400	1	-6
7	8400	8400	1	-6
8	8400	8400	1	-6
9	8400	8400	1	-6
Average	8400	8400	1	

The minimum value of the objective function is $z = -6$. This value was reached 10 out of 10 times.

6.5.7 Problem 7

Minimize

$$z = (y_1 - 1.0)^2 + (y_2 - 2.0)^2 + (y_3 - 1.0)^2 - \ln(y_4 + 1.0) + (x_1 - 1.0)^2 + (x_2 - 2.0)^2 + (x_3 - 3.0)^2$$

Subject to

$$\begin{aligned}
 y_1 + y_2 + y_3 + x_1 + x_2 + x_3 &\leq 5.0 \\
 y_2^2 + x_1^2 + x_2^2 + x_3^2 &\leq 5.5 \\
 y_1 + x_1 &\leq 1.2 \\
 y_2 + x_2 &\leq 1.8 \\
 y_3 + x_3 &\leq 2.5 \\
 y_4 + x_1 &\leq 1.2 \\
 y_2^2 + x_2^2 &\leq 1.64 \\
 y_3^2 + x_3^2 &\leq 4.25 \\
 y_2^2 + x_3^2 &\leq 4.64 \\
 x &\geq 0 \\
 y &= \{0, 1\}^4
 \end{aligned}$$

The results of ten runs of the algorithm are summarized in the next table.

Run	Function Evaluation	Constraint Evaluations	Iterations	Objective Value
0	1445010	13005090	13682	5.63273
1	1433670	12903030	13574	5.579582
2	1413195	12718755	13379	4.579582
3	1337700	12039300	12660	4.579582
4	1393980	12545820	13196	4.579582
5	1475670	13281030	13974	5.63273
6	1352820	12175380	12804	4.579582
7	1507380	13566420	14276	5.27273
8	1430835	12877515	13547	5.579582
9	1330350	11973150	12590	5.27273
Average	1412061	12708549	13368.2	

The minimum value of the objective function is $z = 4.579582$. This value was reached 4 out of 10 times.

6.5.8 Problem 8

Maximise

$$z = r_1 * r_2 * r_3$$

Subject to

$$\begin{aligned}
 r_1 &= 1.0 - 0.1^{y_1} * 0.2^{y_2} * 0.15^{y_3} \\
 r_2 &= 1.0 - 0.05^{y_4} * 0.2^{y_5} * 0.15^{y_6} \\
 r_3 &= 1.0 - 0.02^{y_7} * 0.06^{y_8} \\
 y_1 + y_2 + y_3 &\geq 1 \\
 y_4 + y_5 + y_6 &\geq 1 \\
 y_7 + y_8 &\geq 1 \\
 3.0y_1 + y_2 + 2.0y_3 + 3.0y_4 + 2.0y_5 + y_6 + 3.0y_7 + 2.0y_8 &\leq 10 \\
 y &= \{0, 1\}^8
 \end{aligned}$$

The results of ten runs of the algorithm are summarized in the next table.

Run	Function Evaluation	Constraint Evaluations	Iterations	Objective Value
0	613725	4200420	10000	-0.9507738
1	609000	4200420	10000	-0.9502312
2	593250	4200420	10000	-0.9503424
3	603225	4200420	10000	-0.9504514
4	612150	4200420	10000	-0.9506927
5	586425	4200420	10000	-0.9513166
6	602700	4200420	10000	-0.9502679
7	603750	4200420	10000	-0.9508317
8	618975	4200420	10000	-0.9501467
9	634200	4200420	10000	-0.9506071
Average	607740	4200420	10000	

The minimum value of the objective function is $z = -0.93634$. This value was never reached.

6.6 Conclusions

1. Does the algorithm actually work? Does it successfully find the global optimum of test functions and simple linear programming problems?

We see that the algorithm faired well optimizing the test problems. The linear programming produced mixed results with the first problem reaching the optimum in each case, the second problem reaching the optimum in 70% of the cases while the third problem never reaches the global optimum.

2. Does the algorithm find the global optimum of mixed integer non linear programming problems we chose to test it against?

The mixed integer non linear programming problems presented also produced mixed results. Some of the problems were solved in all 10 runs of the algorithm. Others did not even come close to the global optimum.

Chapter 7

Conclusion

7.1 Equality Constraints

Equality constraints present a problem in the penalty function. An inequality constraint is satisfied on at least one side of the hyperplane defined by the constraint.

An equality constraint is never satisfied as long as the solution does not lie exactly on the hyperplane defined by the result. Thus the penalty function always includes this error, even when a solution may lie very close to the constraint.

This situation can occur frequently in evolutionary strategies where we make random jumps in each object variable. One solution may lie very close to the equality constraint, but a mutation may force it further away. Feasibility decisions are also adversely affected. A solution may be reasonably close to the optimal solution, but is rejected due to not being feasible because of the equality constraint.

Possible heuristics include:

Define a tolerance region around the equality constraint where no penalty is incurred when a solution lies within this region. Shrink this region as the evolution progresses.

Consider the equality constraint $h(\mathbf{x}) = 0$. Suppose we used a quadratic penalty function that adds the square of value of $h(x)$ to the penalty value. The tolerance region around this constraint then stipulates that we don't add to the penalty value for this member if $-t \leq h(\mathbf{x}) \leq t$, where t is the nonnegative tolerance value for this constraint. Let $t \rightarrow 0$ as the number of generations increase.

7.2 Divergence

Running the test functions utilizing correlated mutations, unbounded variables with approximately 30 variables, the objective value grew without bound.

When we are far away from the optimal solution the step size increases until the entire solution space is spanned by one step. We are then reduced to taking giant random steps around the optimal solution.

7.3 Conclusion

Since we don't have a definitive characterization of the optimal solution that can be used by the EA as termination criterion (such as the KKT optimality conditions for linear programming) in MINLP, multiple runs may be necessary to achieve a measure of confidence that we possess a good solution. This may or may not be the global optimum solution nor may it be even close to the optimal solution. We can however compare the solutions found with previous runs of the algorithm to judge whether the new solution is an improvement over previous known solutions or not.

That is if we find any type of solution at all. The experiments described in this paper is hardly conclusive for either the success or failure of the standard EA for the solution of MINLP problems, but it shows that (1) ES have the potential to solve typical MINLP problems, and (2) their success probability depends very much on the problem-structure and dimensionality.

Further work will be needed to make ES a more robust solver in this problem domain and to figure out the essential differences between problems that are easy to be solved with ES and difficult. The results of the current study may serve as a good starting point for this research.

Chapter 8

Further Work

Considering the fact that the results of our experiments are far from conclusive, there are a lot of research still to be done. This section highlights a few points we think are worth looking at:

8.1 Generating feasible solutions vs. Penalty function

During the evolution, force the operators to generate feasible solutions instead of using a penalty function. This can be accomplished by setting an overstepped bound equal to the bound or by reflecting the overstep back into the feasible region.

Our experience has shown that modifying object variables has an adverse effect on the evolution process

8.2 Penalty function optimization

The penalty function is an important part of the constrained evolution process. This function can be optimized by prioritising different penalties. Bounds may be hard or soft, Constraints may be hard or soft. Solutions closer to the feasible region may be less severely penalized. An ES may be used to discover the weights for each penalty variable.

Keeping all this in mind, the penalty function evaluation should be cheap since it is done once for each member of the expanded population for each generation

8.3 Coupling the discrete and continuous vector optimization

This is the central point in MINLP problem solutions. How do we optimize both vectors simultaneously?

The algorithm in this paper attempts to optimize a continuous constraint NLP for every discrete configuration we attempt. This is extremely inefficient

since the current discrete configuration may be very weak, leaving the continuous optimization doing a lot of unnecessary work solving for the optimal solution given the discrete configuration.

The coupling between these optimizations is critical to the general performance of the method.

Chapter 9

Code

9.1 util.h

```
#include <float.h>
#include <math.h>
#include <stdlib.h>
#include <stdio.h>

int dblcomp(double a, double b) {

    double diff = a - b;
    int result;

    if (fabs(diff) < DBL_EPSILON) {
        result = 0;
    } else if (diff <= -DBL_EPSILON) {
        result = -1;
    } else if (diff >= DBL_EPSILON) {
        result = 1;
    } else {
        printf("a = %e\tb = %e\ta - b = %e\n", a, b, a - b);
        exit(EXIT_FAILURE);
    }

    return result;
}

void error(char *error_text, char *file, int line) {
    fprintf(stderr, "%s:%d\n", file, line);
    fprintf(stderr, "Run-time error...\n");
    fprintf(stderr, "%s\n", error_text);
    fprintf(stderr, "...now exiting to system...\n");
    exit(EXIT_FAILURE);
}
```

9.2 rand.h

```
#ifndef RAND_H
#define RAND_H

// Random number generator
double rand2(void);

// Seed the random number generator
void srand2(long seed);

// returns a random integer in the range [0, RAND_MAX]
int rand2i(void);

// returns a random integer in the range [lower, upper]
int rand2ib(int lower, int upper);

// returns a normal random variable with mean mean and standard deviation std
double normal(double mean, double std);

#endif // RAND_H
```

9.3 rand.c

```
#include <stdlib.h>
#include <math.h>
#include <float.h>
#include "rand.h"

// private function prototypes
static double ran2(long *idum);
static double gasdev(long *idum);

// ran2 defines from numerical recipes
#define IM1 2147483563
#define IM2 2147483399
#define AM (1.0 / IM1)
#define IMM1 (IM1 - 1)
#define IA1 40014
#define IA2 40692
#define IQ1 53668
#define IQ2 52774
#define IR1 12211
#define IR2 3791
#define NTAB 32
#define NDIV (1 + IMM1 / NTAB)
#define EPS DBL_EPSILON
#define RNMX (1.0 - EPS)
static long ran2seed = 1;
```

```
// end ran2() defines

// wrapper for ran2, returns double in the range (0.0, 1.0)
double rand2(void) {
    return ran2(&ran2seed);
}

// wrapper for ran2, returns int in the range [0, RAND_MAX]
int rand2i(void) {
    int result;

    result = (int)(((double)RAND_MAX + 1.0) * rand2());

    return result;
}

// wrapper for ran2m returns int in the range [lower, upper]
int rand2ib(int lower, int upper) {

    int result;

    result = lower + (int)(((double)(upper - lower) + 1.0) * rand2());

    return result;
}

// replacement for srand()
void srand2(long seed) {
    if (seed >= 0) {
        seed = -seed;
    }

    ran2seed = seed;
    // splint complains about ignored return value, cast to void to indicate
    // this is deliberate
    (void)ran2(&ran2seed);
}

// replacement for rand()
double ran2(long *idum) {

    long j;
    long k;
    static long idum2 = 123456789;
    static long iy = 0;
    static long iv[NTAB];
    double temp;

    if (*idum <= 0) {
        if (-(*idum) < 1) {
```

```

        *idum = 1;
    } else {
        *idum = -(*idum);
    }

    idum2 = (*idum);

    for (j = NTAB + 7; j >= 0; j--) {
        k = (*idum) / IQ1;
        *idum = IA1 * (*idum - k * IQ1) - k * IR1;

        if (*idum < 0) {
            *idum += IM1;
        }

        if (j < NTAB) {
            iv[j] = *idum;
        }
    }
    iy = iv[0];
}

k = (*idum) / IQ1;
*idum = IA1 * (*idum - k * IQ1) - k * IR1;

if (*idum < 0) {
    *idum += IM1;
}

k = idum2 / IQ2;
idum2 = IA2 * (idum2 - k * IQ2) - k * IR2;

if (idum2 < 0) {
    idum2 += IM2;
}

j = iy / NDIV;
iy = iv[j] - idum2;
iv[j] = *idum;

if (iy < 1) {
    iy += IMM1;
}

temp = AM * iy;

if ((temp - 1.0) > DBL_EPSILON) {
    return RNMX;
} else {
    return temp;
}

```

```

    }
}

// returns a N(mean, std) distributed number
double normal(double mean, double std) {
    double result;

    result = (gasdev(&ran2seed) * std) + mean;

    return result;
}

// generates normal(0,1) distributed numbers
double gasdev(long *idum) {

    static int iset = 0;
    static double gset;
    double fac, rsq, v1, v2;

    if (*idum < 0) {
        iset = 0;
    }

    if (iset == 0) {
        do {
            v1 = 2.0 * ran2(idum) - 1.0;
            v2 = 2.0 * ran2(idum) - 1.0;
            rsq = v1 * v1 + v2 * v2;
        } while (rsq >= 1.0 || rsq < DBL_EPSILON);

        fac = sqrt(-2.0 * log(rsq) / rsq);

        gset = v1 * fac;
        iset = 1;
        return v2 * fac;
    } else {
        iset = 0;
        return gset;
    }
}

```

9.4 poputils.h

```

#ifndef POPUTILS_H
#define POPUTILS_H

#include <stdlib.h>
#include "algo.h"

```

```

member *alloc_pop(size_t popsize);
void free_pop(member *pop, size_t popsize);
void init_pop(member *pop, size_t popsize);
void printPop(member* pop, size_t popsize, FILE *file);
void eval_pop(member *pop, size_t popsize, double (*obj)(double *values));
void eval_constr(member *pop, size_t popsize,
                double (*con)(double *values, int num));
void eval_bounds(member *pop, size_t popsize);
int getBest(member *pop, size_t popsize);
int getBestAvail(member *pop, int popsize, int *avail);

#endif // POPUTILS_H

```

9.5 poputils.c

```

#include <stdlib.h>
#include <math.h>
#include <stdio.h>
#include "algo.h"
#include "rand.h"
#include "poputils.h"
#include "memutils.h"

extern sigma_t sigma;
extern select_t select_type;
extern size_t ncvvars;
extern size_t ndvars;
extern size_t nconstr;
extern size_t neq_constr;
extern double *lb;
extern double *ub;
extern int linpen;

int getBest(member *pop, size_t popsize) {
    int best = 0;
    size_t i;

    for (i = 1; i < popsize; i++) {
        if (compare(pop + i, pop + best) == 1) {
            best = (int)i;
        }
    }

    return best;
}

int getBestAvail(member *pop, int popsize, int *avail) {
    int best;
    int i;

```

```

    for (best = 0; best < popsize; best++) {
        if (*(avail + best) == 1) {
            break;
        }
    }

    for (i = best + 1; i < popsize; i++) {
        if (*(avail + i) == 1) {
            if (compare(pop + i, pop + best) == 1) {
                best = i;
            }
        }
    }

    return best;
}

void eval_bounds(member *pop, size_t popsize) {
    size_t i, j;
    member *mem;
    double result = 0;

    for (i = 0; i < popsize; i++) {
        mem = pop + i;

        for (j = 0; j < ncvvars; j++) {
            if (dblcomp(*(mem->cvars + j), ub[j]) == 1) {
                if (linpen == 1) {
                    result += (*(mem->cvars + j) - ub[j]);
                } else {
                    result += pow(*(mem->cvars + j) - ub[j], 2.0);
                }
            } else if (dblcomp(*(mem->cvars + j), lb[j]) == -1) {
                if (linpen == 1) {
                    result += (lb[j] - *(mem->cvars + j));
                } else {
                    result += pow(lb[j] - *(mem->cvars + j), 2.0);
                }
            }
        }
        mem->bpen = result;
        result = 0.0;
    }
}

void eval_constr(member *pop, size_t popsize, double (*con)(double *values, int num)) {
    size_t i, j;
    double eq = 0;
    double neq = 0;

```

```

member *mem;
double *values = (double*)calloc((ncvars + ndvars), sizeof(double));

for (i = 0; i < popsize; i++) {
    mem = pop + i;

    for (j = 0; j < ncvars; j++) {
        *(values + j) = *(mem->cvars + j);
    }
    for (j = ncvars; j < ncvars + ndvars; j++) {
        *(values + j) = (double)*(mem->dvars + j));
    }

    for (j = 0; j < neq_constr; j++) {
        if (linpen == 1) {
            eq += fabs(con(values, (int)j));
        } else {
            eq += pow(con(values, (int)j), 2.0);
        }
    }
    mem->eq_cpen = eq;

    if (mem->valid == -1) {
        if (isnan(mem->eq_cpen) != 0) {
            mem->valid = 0;
        } else {
            mem->valid = 1;
        }
    } else if (mem->valid == 1) {
        if (isnan(mem->eq_cpen) != 0) {
            mem->valid = 0;
        }
    }
}

eq = 0;

for (j = neq_constr; j < nconstr; j++) {
    if (linpen == 1) {
        neq += fabs(con(values, (int)j));
    } else {
        neq += pow(con(values, (int)j), 2.0);
    }
}
mem->cpen = neq;
if (mem->valid == -1) {
    if (isnan(mem->cpen) != 0) {
        mem->valid = 0;
    } else {
        mem->valid = 1;
    }
}

```

```

        } else if (mem->valid == 1) {
            if (isnan(mem->cpen) != 0) {
                mem->valid = 0;
            }
        }
        neq = 0;
    }
    free(values);
}

void eval_pop(member *pop, size_t popsize, double (*obj)(double *values)) {
    size_t i, j;
    double *values = (double*)calloc(ncvars + ndvars, sizeof(double));
    member *mem;

    for (i = 0; i < popsize; i++) {
        mem = pop + i;

        for (j = 0; j < ncvars; j++) {
            *(values + j) = *(mem->cvars + j);
        }
        for (j = ncvars; j < ncvars + ndvars; j++) {
            *(values + j) = (double)*(mem->dvars + j);
        }

        mem->obj = obj(values);

        if (mem->valid == -1) {
            if (isnan(mem->obj) != 0) {
                mem->valid = 0;
            } else {
                mem->valid = 1;
            }
        } else if (mem->valid == 1) {
            if (isnan(mem->obj) != 0) {
                mem->valid = 0;
            }
        }
    }

    free(values);
}

void printPop(member* pop, size_t popsize, FILE *file) {

    size_t i, j;
    member *mem;

    if (file == NULL) {

```

```

        file = stdout;
    }

    for (i = 0; i < popsize; i++) {
        mem = pop + i;

        for (j = 0; j < 80; j++) {
            fprintf(file, "=");
        }
        fprintf(file, "\n");

        printMember(mem, file);
    }

    for (j = 0; j < 80; j++) {
        fprintf(file, "=");
    }
    fprintf(file, "\n");
}

void init_pop(member *pop, size_t popsize) {
    size_t i, j;
    member *mem;
    size_t sigmavars = 0;
    size_t alphavars = 0;

    if (sigma == SINGLE) {
        sigmavars = 1;
        alphavars = 0;
    } else if (sigma == MULTIPLE) {
        sigmavars = ncvars;
        alphavars = 0;
    } else if (sigma == CORRELATED) {
        sigmavars = ncvars;
        alphavars = ncvars * (ncvars - 1) / 2;
    }

    for (i = 0; i < popsize; i++) {
        mem = pop + i;

        if (ncvars > 0) {
            for (j = 0; j < ncvars; j++) {
                *(mem->cvars + j) = lb[j] + (rand2() * (ub[j] - lb[j]));
            }
            for (j = 0; j < sigmavars; j++) {
                *(mem->sigma + j) = (ub[j] - lb[j]) / 6.0;
            }
            if (alphavars > 0) {
                for (j = 0; j < alphavars; j++) {
                    *(mem->alpha + j) = (rand2() * (2.0 * M_PI)) - M_PI;
                }
            }
        }
    }
}

```

```

        }
    }
}

if (ndvars > 0) {
    for (j = 0; j < ndvars; j++) {
        *(mem->dvars + j) = rand2ib((int)lb[ncvars + j], (int)ub[ncvars + j]);
    }
}

mem->obj = 0.0;
mem->cpen = 0.0;
mem->bpen = 0.0;
mem->eq_cpen = 0.0;
mem->age = 0;
mem->valid = -1;
}
}

void free_pop(member *pop, size_t popsize) {
    size_t i;
    member *mem;

    for (i = 0; i < popsize; i++) {
        mem = pop + i;

        if (mem->cvars != NULL) {
            free(mem->cvars);
            mem->cvars = NULL;
            free(mem->sigma);
            mem->sigma = NULL;
            if (mem->alpha != NULL) {
                free(mem->alpha);
                mem->alpha = NULL;
            }
        }

        if (mem->dvars != NULL) {
            free(mem->dvars);
            mem->dvars = NULL;
        }
    }

    free(pop);
}

member *alloc_pop(size_t popsize) {
    member *result;
    member *mem;
    size_t i;

```

```

size_t sigmavars = 0;
size_t alphavars = 0;

result = (member*)calloc(popsize, sizeof(member));
if (result == NULL) {
    error("Failed to allocate storage", __FILE__, __LINE__);
}

if (sigma == SINGLE) {
    sigmavars = 1;
    alphavars = 0;
} else if (sigma == MULTIPLE) {
    sigmavars = ncvars;
    alphavars = 0;
} else if (sigma == CORRELATED) {
    sigmavars = ncvars;
    alphavars = ncvars * (ncvars - 1) / 2;
}

for (i = 0; i < popsize; i++) {
    mem = result + i;

    if (ncvars == 0) {
        mem->cvars = NULL;
        mem->sigma = NULL;
        mem->alpha = NULL;
    } else {
        mem->cvars = (double*)calloc(ncvars, sizeof(double));
        mem->sigma = (double*)calloc(sigmavars, sizeof(double));
        if ((mem->cvars == NULL) || (mem->sigma == NULL)) {
            error("Failed to allocate storage", __FILE__, __LINE__);
        }
        if (alphavars > 0) {
            mem->alpha = (double*)calloc(alphavars, sizeof(double));
            if (mem->alpha == NULL) {
                error("Failed to allocate storage", __FILE__, __LINE__);
            }
        } else {
            mem->alpha = NULL;
        }
    }
}

if (ndvars == 0) {
    mem->dvars = NULL;
} else {
    mem->dvars = (int*)calloc((size_t)ndvars, sizeof(int));
    if (mem->dvars == NULL) {
        error("Failed to allocate storage", __FILE__, __LINE__);
    }
}
}

```

```

    }

    return result;
}

```

9.6 operators.h

```

#ifndef OPERATORS_H
#define OPERATORS_H

void recombine_c(member *pop, member *offspring, size_t popsize, size_t offsize);
void mutate_c(member *pop, size_t popsize);
void selection(member *pop, member *offspring, size_t npop, size_t noff);

#endif // OPERATORS_H

```

9.7 operators.c

```

#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <float.h>
#include "algo.h"
#include "rand.h"
#include "util.h"
#include "poputils.h"

extern size_t ncvars;
extern size_t ndvars;
extern sigma_t sigma;
extern select_t select_type;
extern int maxage;

static void mutate_single(member *pop, size_t popsize);
static void mutate_multiple(member *pop, size_t popsize);
static void mutate_correlated(member *pop, size_t popsize);
static void select_top(member *pop, member *offspring, size_t npop, size_t noff);
static void select_topplus(member *pop, member *offspring, size_t npop, size_t noff);

// testing
double length(double *val, int len) {
    double sum = 0.0;
    int i;

    for (i = 0; i < len; i++) {
        sum += val[i] * val[i];
    }
}

```

```

    return sqrt(sum);
}

void fixSigmas(member *mem) {
    int j;
    double xlen;
    double slen;

    xlen = length(mem->cvars, ncvars);
    slen = length(mem->sigma, ncvars);

    if (dblcomp(xlen * 0.9, slen) <= 0) {
        for (j = 0; j < ncvars; j++) {
            *(mem->sigma + j) /= 2.0;
        }
    }
}

// ~testing

void selection(member *pop, member *offspring, size_t npop, size_t noff) {
    switch(select_type) {
        case TOP:
            select_top(pop, offspring, npop, noff);
            break;
        case TOPPLUS:
            select_topplus(pop, offspring, npop, noff);
            break;
        case MKL:
            select_topplus(pop, offspring, npop, noff);
            break;
    }
}

void select_topplus(member *pop, member *offspring, size_t npop, size_t noff) {
    size_t i;
    int availco = 0;
    int availcp = 0;
    int *availp = (int*)malloc(sizeof(int) * npop);
    int *availo = (int*)malloc(sizeof(int) * noff);
    member *tmppop = alloc_pop(npop);
    member *mem;
    int index = 0;
    int bestp;
    int besto;

    for (i = 0; i < noff; i++) {
        mem = offspring + i;

        if (mem->valid == 1) {

```

```

        if (select_type == MKL) {
            if (mem->age < maxage) {
                *(availo + i) = 1;
                availco++;
            } else {
                *(availo + i) = 0;
            }
        } else {
            *(availo + i) = 1;
            availco++;
        }
    } else {
        *(availo + i) = 0;
    }
}

for (i = 0; i < npop; i++) {
    copyMember(pop + i, tmppop + i);
    mem = tmppop + i;

    if (mem->valid == 1) {
        if (select_type == MKL) {
            if (mem->age < maxage) {
                *(availp + i) = 1;
                availcp++;
            } else {
                *(availp + i) = 0;
            }
        } else {
            *(availp + i) = 1;
            availcp++;
        }
    } else {
        *(availp + i) = 0;
    }
}

if ((availco + availcp) <= npop) {
    for (i = 0; i < npop; i++) {
        mem = tmppop + i;

        if (*(availp + i) == 1) {
            copyMember(mem, pop + index);
            if (select_type == MKL)
                (pop + index)->age += 1;
            index++;
        }
    }

    for (i = 0; i < noff; i++) {

```

```

        mem = offspring + i;

        if (*(availo + i) == 1) {
            copyMember(mem, pop + index);
            if (select_type == MKL)
                (pop + index)->age += 1;
            index++;
        }
    }

    if (index < npop) {
        init_pop(pop + index, npop - index);
    }
} else {
    for (i = 0; i < npop; i++) {
        besto = getBestAvail(offspring, noff, availo);
        bestp = getBestAvail(tmppop, npop, availp);

        if (compare(tmppop + bestp, offspring + besto) == 1) {
            copyMember(tmppop + bestp, pop + i);
            *(availp + bestp) = 0;
        } else {
            copyMember(offspring + besto, pop + i);
            *(availo + besto) = 0;
        }
        if (select_type == MKL)
            (pop + i)->age += 1;
    }
}

free(availo);
free(availp);
free_pop(tmppop, npop);
}

void select_top(member *pop, member *offspring, size_t npop, size_t noff) {

    size_t i;
    int availc = 0;
    int *avail = (int*)malloc(sizeof(int) * noff);
    member *mem;
    int index = 0;

    for (i = 0; i < noff; i++) {
        mem = offspring + i;

        if (mem->valid == 1) {
            *(avail + i) = 1;
            availc++;
        } else {

```

```

        *(avail + i) = 0;
    }
}

if (availc <= npop) {
    for (i = 0; i < noff; i++) {
        mem = offspring + i;

        if (*(avail + i) == 1) {
            copyMember(mem, pop + index);
            index++;
        }
    }

    if (index < npop) {
        init_pop(pop + index, npop - index);
    }
} else {
    for (i = 0; i < npop; i++) {
        index = getBestAvail(offspring, noff, avail);
        copyMember(offspring + index, pop + i);
        *(avail + index) = 0;
    }
}

free(avail);
}

void mutate_c(member *pop, size_t popsize) {
    switch(sigma) {
        case SINGLE:
            mutate_single(pop, popsize);
            break;
        case MULTIPLE:
            mutate_multiple(pop, popsize);
            break;
        case CORRELATED:
            mutate_correlated(pop, popsize);
            break;
    }
}

void mutate_single(member *pop, size_t popsize) {
    size_t i, j;
    double tau = 1.0 / sqrt(ncvars);
    member *mem;

    for (i = 0; i < popsize; i++) {
        mem = pop + i;
        *(mem->sigma) *= exp(tau * normal(0.0, 1.0));
    }
}

```

```

        if (dblcomp(*(mem->sigma), DBL_EPSILON) <= 0) {
            *(mem->sigma) = DBL_EPSILON;
        }

        for (j = 0; j < ncvars; j++) {
            *(mem->cvars + j) += *(mem->sigma) * normal(0.0, 1.0));
        }
    }
}

void mutate_multiple(member *pop, size_t popsize) {
    double tau = 1.0 / sqrt(2.0 * sqrt(ncvars));
    double taup = 1.0 / sqrt(2.0 * ncvars);
    size_t i, j;
    member *mem;

    for (i = 0; i < popsize; i++) {
        mem = pop + i;
        for (j = 0; j < ncvars; j++) {
            *(mem->sigma + j) *= exp(taup * normal(0.0, 1.0) +
                                   tau * normal(0.0, 1.0));

            if (dblcomp(*(mem->sigma + j), DBL_EPSILON) <= 0) {
                *(mem->sigma + j) = DBL_EPSILON;
            }

            *(mem->cvars + j) += *(mem->sigma + j) * normal(0.0, 1.0));
        }
    }
}

void mutate_correlated(member *pop, size_t popsize) {
    double tau = 1.0 / sqrt(2.0 * sqrt(ncvars));
    double taup = 1.0 / sqrt(2.0 * ncvars);
    double beta = (M_PI * 5) / 180.0;
    size_t i, j, k;
    member *mem;
    size_t avars = ncvars * (ncvars - 1) / 2;
    size_t nq;
    double *sigmau = (double*)calloc(ncvars, sizeof(double));
    size_t n1, n2;
    double d1, d2;
    double aalpha;

    for (i = 0; i < popsize; i++) {
        mem = pop + i;
        for (j = 0; j < ncvars; j++) {
            *(mem->sigma + j) *= exp(taup * normal(0.0, 1.0) +
                                   tau * normal(0.0, 1.0));
        }
    }
}

```

```

        if (dblcomp(*(mem->sigma + j), DBL_EPSILON) <= 0) {
            *(mem->sigma + j) = DBL_EPSILON;
        }
    }

    fixSigmas(mem);

    for (j = 0; j < avars; j++) {
        *(mem->alpha + j) += (beta * normal(0.0, 1.0));

        if (dblcomp(*(mem->alpha + j), M_PI) == 1) {
            *(mem->alpha + j) -= (2.0 * M_PI);
        } else if (dblcomp(*(mem->alpha + j), -M_PI) == -1) {
            *(mem->alpha + j) += (2.0 * M_PI);
        }
    }

    for (j = 0; j < ncvars; j++) {
        *(sigmau + j) = *(mem->sigma + j) * normal(0.0, 1.0);
    }

    nq = avars - 1;
    for (j = 1; j <= ncvars - 1; j++) {
        n1 = ncvars - j - 1;
        n2 = ncvars - 1;
        for (k = 1; k <= j; k++) {
            aalpha = *(mem->alpha + nq);
            d1 = *(sigmau + n1);
            d2 = *(sigmau + n2);
            *(sigmau + n2) = d1 * sin(aalpha) + d2 * cos(aalpha);
            *(sigmau + n1) = d1 * cos(aalpha) - d2 * sin(aalpha);
            n2 = n2 - 1;
            nq = nq - 1;
        }
    }

    for (j = 0; j < ncvars; j++) {
        *(mem->cvars + j) += normal(0.0, *(sigmau + j));
    }
}

free(sigmau);
}

void recombine_c(member *pop, member *offspring, size_t popsize, size_t offsize) {

    size_t i;
    size_t j;
    int par1;

```

```

int par2;
member *offmem;
member *par1mem;
member *par2mem;

size_t svars;
size_t avars;

if (sigma == SINGLE) {
    svars = 1;
    avars = 0;
} else if (sigma == MULTIPLE) {
    svars = ncvars;
    avars = 0;
} else if (sigma == CORRELATED) {
    svars = ncvars;
    avars = ncvars * (ncvars - 1) / 2;
}

for (i = 0; i < offsize; i++) {
    offmem = offspring + i;

    par1 = rand2ib(0, popsize - 1);
    do {
        par2 = rand2ib(0, popsize - 1);
    } while (par1 == par2);

    par1mem = pop + par1;
    par2mem = pop + par2;

    if (ncvars > 0) {
        for (j = 0; j < ncvars; j++) {
            if (rand2ib(0, 1) == 1) {
                *(offmem->cvars + j) = *(par1mem->cvars + j);
            } else {
                *(offmem->cvars + j) = *(par2mem->cvars + j);
            }
        }

        for (j = 0; j < svars; j++) {
            *(offmem->sigma + j) = (*(par1mem->sigma + j) +
                *(par2mem->sigma + j)) / 2.0;
        }

        if (avars > 0) {
            for (j = 0; j < avars; j++) {
                *(offmem->alpha + j) = (*(par1mem->alpha + j) +
                    *(par2mem->alpha + j)) / 2.0;
            }
        } else {

```

```

        offmem->alpha = NULL;
    }
} else {
    offmem->cvars = NULL;
    offmem->sigma = NULL;
    offmem->alpha = NULL;
}

if (ndvars > 0) {
    if (rand2ib(0, 1) == 1) {
        for (j = 0; j < ndvars; j++) {
            *(offmem->dvars + j) = *(par1mem->dvars + j);
        }
    } else {
        for (j = 0; j < ndvars; j++) {
            *(offmem->dvars + j) = *(par2mem->dvars + j);
        }
    }
} else {
    offmem->dvars = NULL;
}

offmem->obj = 0.0;
offmem->cpen = 0.0;
offmem->bpen = 0.0;
offmem->eq_cpen = 0.0;
offmem->age = 0;
offmem->valid = -1;
}
}

```

9.8 memutils.h

```

#ifndef MEMUTILS_H
#define MEMUTILS_H

#include "algo.h"

void copyMember(member* orig, member* copy);
int compare(member *a, member *b);
void printMember(member *mem, FILE *file);

#endif // MEMUTILS_H

```

9.9 memutils.c

```

#include <stdlib.h>
#include <stdio.h>

```

```

#include "algo.h"

extern size_t ncvars;
extern sigma_t sigma;
extern select_t select_type;
extern size_t ndvars;

static double get_alpha(member* mem, size_t i, size_t j);
static void set_alpha(member* mem, size_t i, size_t j, double value);

void printMember(member *mem, FILE *file) {

    size_t i, j;
    int printed;

    if (file == NULL) {
        file = stdout;
    }

    if (ncvars > 0) {
        fprintf(file, "-----\n");
        fprintf(file, "Continuous\n");
        fprintf(file, "-----\n");
        for (i = 0; i < ncvars; i++) {
            fprintf(file, "x%02d = %e\t", (int)i, *(mem->cvars + i));
            if (((i + 1) % 5 == 0) && ((i + 1) < ncvars)) {
                fprintf(file, "\n");
            }
        }
        fprintf(file, "\n\n");
    }

    if (ndvars > 0) {
        fprintf(file, "-----\n");
        fprintf(file, "Discrete\n");
        fprintf(file, "-----\n");
        for (i = 0; i < ndvars; i++) {
            fprintf(file, "y%02d = %d\t", (int)i, *(mem->dvars + i));
            if (((i + 1) % 5 == 0) && ((i + 1) < ndvars)) {
                fprintf(file, "\n");
            }
        }
        fprintf(file, "\n\n");
    }

    if (ncvars > 0) {
        fprintf(file, "-----\n");
        fprintf(file, "Sigma\n");
        fprintf(file, "-----\n");
    }
}

```

```

    if (sigma == SINGLE) {
        fprintf(file, "s = %+e", *(mem->sigma));
        fprintf(file, "\n\n");
    } else {
        for (i = 0; i < ncvars; i++) {
            fprintf(file, "s%02d = %+e\t", (int)i, *(mem->sigma + i));
            if (((i + 1) % 5 == 0) && ((i + 1) < ncvars)) {
                fprintf(file, "\n");
            }
        }
        fprintf(file, "\n\n");

        if (sigma == CORRELATED) {
            fprintf(file, "-----\n");
            fprintf(file, "Alpha\n");
            fprintf(file, "-----\n");

            printed = 0;

            for (i = 0; i < ncvars - 1; i++) {
                for (j = i + 1; j < ncvars; j++) {
                    fprintf(file, "a(%02d, %02d) = %+e\t", (int)i,
                        (int)j, get_alpha(mem, i, j));
                    printed++;
                    if ((printed % 5 == 0) &&
                        ((printed + 1) <
                         (int)(ncvars * (ncvars - 1) / 2))) {
                        fprintf(file, "\n");
                    }
                }
            }
            fprintf(file, "\n\n");
        }
    }
}

if (select_type == MKL) {
    fprintf(file, "---\n");
    fprintf(file, "Age\n");
    fprintf(file, "---\n");
    fprintf(file, "age = %d", mem->age);
    fprintf(file, "\n\n");
}

fprintf(file, "-----\n");
fprintf(file, "Penalty\n");
fprintf(file, "-----\n");
fprintf(file, "Constraints: %e\n", mem->cpen);
fprintf(file, "Equality Constraints: %e\n", mem->eq_cpen);
fprintf(file, "Bounds: %e", mem->bpen);

```

```

    fprintf(file, "\n\n");

    fprintf(file, "-----\n");
    fprintf(file, "Evaluation\n");
    fprintf(file, "-----\n");
    fprintf(file, "z = %e\n\n", mem->obj);
}

// returns 1 if a < b
//      0 if a > b
//     -1 if a and b are both invalid
int compare(member *a, member *b) {
    int afeas, bfeas;
    int result = 0;
    double apen = a->cpen + a->bpen + a->eq_cpen;
    double bpen = b->cpen + b->bpen + b->eq_cpen;

    if (a->valid <= 0) {
        afeas = -1;
    } else if (dblcomp(apen, 0.0) == 1) {
        afeas = 0;
    } else {
        afeas = 1;
    }

    if (b->valid <= 0) {
        bfeas = -1;
    } else if (dblcomp(bpen, 0.0) == 1) {
        bfeas = 0;
    } else {
        bfeas = 1;
    }

    if ((afeas == -1) && (bfeas == -1)) {
        result = -1;
    } else if ((afeas == -1) && (bfeas == 0)) {
        result = 0;
    } else if ((afeas == -1) && (bfeas == 1)) {
        result = 0;
    } else if ((afeas == 0) && (bfeas == -1)) {
        result = 1;
    } else if ((afeas == 0) && (bfeas == 0)) {
        if (dblcomp(apen, bpen) <= 0) {
            result = 1;
        } else {
            result = 0;
        }
    } else if ((afeas == 0) && (bfeas == 1)) {
        result = 0;
    } else if ((afeas == 1) && (bfeas == -1)) {

```

```

        result = 1;
    } else if ((afeas == 1) && (bfeas == 0)) {
        result = 1;
    } else if ((afeas == 1) && (bfeas == 1)) {
        if (dblcomp(a->obj, b->obj) <= 0) {
            result = 1;
        } else {
            result = 0;
        }
    }
}

return result;
}

void copyMember(member* orig, member* copy) {
    size_t i;
    if (ncvars > 0) {
        for (i = 0; i < ncvars; i++) {
            *(copy->cvars + i) = *(orig->cvars + i);
        }

        if (sigma == SINGLE) {
            *(copy->sigma) = *(orig->sigma);
        } else {
            for (i = 0; i < ncvars; i++) {
                *(copy->sigma + i) = *(orig->sigma + i);
            }

            if (sigma == CORRELATED) {
                for (i = 0; i < (ncvars * (ncvars - 1) / 2); i++) {
                    *(copy->alpha + i) = *(orig->alpha + i);
                }
            } else {
                if (copy->alpha != NULL) {
                    free(copy->alpha);
                }
                copy->alpha = NULL;
            }
        }
    }
} else {
    if (copy->cvars != NULL) {
        free(copy->cvars);
    }
    copy->cvars = NULL;
    if (copy->sigma != NULL) {
        free(copy->sigma);
    }
    copy->sigma = NULL;
    if (copy->alpha != NULL) {
        free(copy->alpha);
    }
}

```

```

    }
    copy->alpha = NULL;
}

if (ndvars > 0) {
    for (i = 0; i < ndvars; i++) {
        *(copy->dvars + i) = *(orig->dvars + i);
    }
} else {
    if (copy->dvars != NULL) {
        free(copy->dvars);
    }
    copy->dvars = NULL;
}

copy->obj = orig->obj;
copy->cpen = orig->cpen;
copy->bpen = orig->bpen;
copy->eq_cpen = orig->eq_cpen;

if (select_type == MKL) {
    copy->age = orig->age;
} else {
    copy->age = 0;
}

copy->valid = orig->valid;
}

double get_alpha(member* mem, size_t i, size_t j) {
    double result;
    size_t index;

    if (i == j) {
        result = 0.0e0;
    } else if (j < i) {
        index = (j * ncvars) - (((j * j) + j)/2) + i - j - 1;
        result = *(mem->alpha + index);
    } else {
        index = (i * ncvars) - (((i * i) + i)/2) + j - i - 1;
        result = *(mem->alpha + index);
    }

    return result;
}

void set_alpha(member* mem, size_t i, size_t j, double value) {
    size_t index;

    if (i == j) {

```

```

        return;
    } else if (j < i) {
        index = (j * ncvvars) - (((j * j) + j)/2) + i - j - 1;
        *(mem->alpha + index) = value;
    } else {
        index = (i * ncvvars) - (((i * i) + i) / 2) + j - i - 1;
        *(mem->alpha + index) = value;
    }
}

```

9.10 algo.h

```

#ifndef ALGO_H
#define ALGO_H

#include <stdlib.h>

typedef struct {
    double *cvars;
    int *dvars;
    double *sigma;
    double *alpha;
    double obj;
    double cpen;
    double bpen;
    double eq_cpen;
    int age;
    int valid;
} member;

typedef enum {SINGLE, MULTIPLE, CORRELATED} sigma_t;
typedef enum {TOP, TOPPLUS, MKL} select_t;

void init(char *basename, char *args, size_t npop, size_t noffspring,
          sigma_t sigmatype, select_t selecttype, size_t contvars,
          size_t discvars, size_t constr, size_t eq_constr, double *lowerb,
          double *upperb, double (*objective)(double *values),
          double (*constraint)(double *values, int num),
          int linear, int max_age);
void algo(void);

#endif // ALGO_H

```

9.11 algo.c

```

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

```

```
#include <stdlib.h>
#include "algo.h"
#include "rand.h"
#include "util.h"
#include "poputils.h"
#include "operators.h"
#include "memutils.h"

static void initRand(void);
static void openFiles(char *basename, char *args);
static void writeParams(FILE *out);
static void cleanup(void);

static size_t mu;
static size_t lamda;
sigma_t sigma;
select_t select_type;
size_t ncvvars;
size_t ndvvars;
size_t nconstr;
size_t neq_constr;
static FILE *flog;
static FILE *fpar;
static member *pop;
static member *offspring;
double *lb;
double *ub;
static double (*fobj)(double *values);
static double (*fcon)(double *values, int num);
int linpen;
int maxage;

void algo(void) {

    int term = 1;
    member *solution = alloc_pop(1);
    int gen = 0;
    int best = getBest(pop, mu);
    copyMember(pop + best, solution);

    while (term == 1) {
        recombine_c(pop, offspring, mu, lamda);
        mutate_c(offspring, lamda);

        eval_pop(offspring, lamda, fobj);
        eval_constr(offspring, lamda, fcon);
        eval_bounds(offspring, lamda);

        selection(pop, offspring, mu, lamda);
        best = getBest(pop, mu);
    }
}
```

```

        if (compare(pop + best, solution) == 1) {
            printf("%04d New Best solution: %e\n", gen, solution->obj);
            copyMember(pop + best, solution);
        }

        if (dblcomp(solution->obj, 0.0) <= 0) {
            term = 0;
            printMember(solution, NULL);
        }

        gen++;
    }

    cleanup();
}

void init(char *basename, char *args, size_t npop, size_t noffspring, sigma_t sigmatype, select

    mu = npop;
    lamda = noffspring;
    sigma = sigmatype;
    select_type = selecttype;
    ncvars = contvars;
    ndvars = discvars;
    nconstr = constr;
    neq_constr = eq_constr;
    lb = lowerb;
    ub = upperb;
    fobj = objective;
    fcon = constraint;
    linpen = linear;
    maxage = max_age;

    initRand();
    openFiles(basename, args);

    pop = alloc_pop(mu);
    offspring = alloc_pop(lamda);
    init_pop(pop, mu);
    eval_pop(pop, mu, fobj);
    eval_constr(pop, mu, fcon);
    eval_bounds(pop, mu);

    writeParams(fpar);
}

void writeParams(FILE *out) {

    if (out == NULL) {

```

```

        out = stdout;
    }

    fprintf(out, "-----\n");
    fprintf(out, "Parameters\n");
    fprintf(out, "-----\n\n");

    fprintf(out, "Population Size: %d\n", mu);
    fprintf(out, "Offspring Size: %d\n", lamda);
    fprintf(out, "Mutation: ");

    switch(sigma) {
        case SINGLE:
            fprintf(out, "Simple, single step size\n");
            break;
        case MULTIPLE:
            fprintf(out, "Simple, multiple step sizes\n");
            break;
        case CORRELATED:
            fprintf(out, "Correlated\n");
            break;
    }

    fprintf(out, "Selection: ");

    switch (select_type) {
        case TOP:
            fprintf(out, "Deterministic rank based, comma\n");
            break;
        case TOPPLUS:
            fprintf(out, "Deterministic rank based, plus\n");
            break;
        case MKL:
            fprintf(out, "Deterministic rank based, plus, with ageing\n");
            fprintf(out, "\tMaximum Age: %d\n", maxage);
            break;
    }

    fprintf(out, "Linear Penalty: %s\n", linpen == 0 ? "False" : "True");

    fprintf(out, "\n");
    fprintf(out, "-----\n");
    fprintf(out, "Problem Parameters\n");
    fprintf(out, "-----\n\n");

    fprintf(out, "Continuous Variables: %d\n", ncvvars);
    fprintf(out, "Discrete Variables: %d\n", ndvvars);
    fprintf(out, "Number of Constraints: %d\n", nconstr);
    fprintf(out, "Number of Equality Constraints: %d\n", neq_constr);

```

```
    fflush(out);
}

void cleanup(void) {
    int val;

    free_pop(pop, mu);
    pop = NULL;
    free_pop(offspring, lamda);
    offspring = NULL;
    val = fclose(fpar);
    val = fclose(flog);
}

void initRand(void) {
    FILE *urand;
    char *filename = "/dev/urandom";
    long rdata;
    size_t freadret;
    int fcloseret;

    urand = fopen(filename, "r");

    if (urand == NULL) {
        error("Failed to open /dev/urandom", __FILE__, __LINE__);
    } else {
        freadret = fread((void*)&rdata, sizeof(long), 1, urand);

        if (freadret < 1) {
            error("Failed to read /dev/urandom", __FILE__, __LINE__);
        }

        fcloseret = fclose(urand);
        if (fcloseret != 0) {
            error("Failed to close /dev/urandom", __FILE__, __LINE__);
        }
        srand2(rdata);
    }
}

void openFiles(char *basename, char *args) {

    size_t flen;
    char *fparname;
    char *flogname;
    int val;
    int num;

    if (strlen(args) == 0) {
        flen = strlen(basename) + 10;
```

```
} else {
    flen = strlen(basename) + strlen(args) + 6;
}

fparname = (char*)calloc(flen, sizeof(char));
flogname = (char*)calloc(flen, sizeof(char));
if ((fparname == NULL) || (flogname == NULL)) {
    error("Failed to allocate memory", __FILE__, __LINE__);
}

if (strlen(args) == 0) {
    num = rand2ib(0, 9999);
    val = snprintf(fparname, flen, "%s-%04d.par", basename, num);
    val = snprintf(flogname, flen, "%s-%04d.log", basename, num);
} else {
    val = snprintf(fparname, flen, "%s-%s.par", basename, args);
    val = snprintf(flogname, flen, "%s-%s.log", basename, args);
}

flog = fopen(flogname, "w");
fpar = fopen(fparname, "w");

free(fparname);
free(flogname);
}
```

Chapter 10

Acknowledgements

I would like to thank the following people and organizations for making my time at Leiden University possible:

1. Michael Emmerich, my project leader. A very patient man.
2. The Zuid Afrika Huis (www.zuidafrikahuis.nl) for their financial support.

Bibliography

- [1] Brian Borchers. (minlp): Branch and bound methods, November 1997. <http://citeseer.ist.psu.edu/155119.html>.
- [2] S.F. de Azevedo D. Barbosa M.F. Cardoso, R.L. Salcedo. A simulated annealing approach to the solution of minlp problems. *Computers and Chemical Engineering*, Volume 21(Number 12):pp 1349 – 1364, September 1997.
- [3] Bernd Gross Martin Schutz Michael Emmerich, Monika Grotzner. Mixed integer evolution strategy for chemical plant optimization with simulators. *In I. C. Parmee: Selected papers from Int. Conference on Adaptive Design and Manufacture, ACDM 2000*, 2000.
- [4] Hanif D Sherali Mokhtar S Bazaraa, John J Jarvis. *Linear Programming and Network Flows*. John Wiley and Sons Inc, second edition edition, 1990.
- [5] Jeroen Eggermont Ernst G.P. Bovenkamp Rui Li, Michael T.M. Emmerich. Mixed-integer optimization of coronary vessel image analysis using evolution strategies. *Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pages 1645 – 1652, 2006.