Lab 4

Dr. Donald Davendra
CS471 - Optimization

November 16, 2016

1 Introduction

The lab introduces another three of the new meta-heuristics. These are the Self-Organising Migrating Algorithm (SOMA), Firefly Algorithm (FA) and Harmony Search Algorithm (HS).

This lab requires you to code the three algorithms in a common framework and conduct experiments on the unimodel and multimodel problems. The algorithms are described in the following sections.

2 Self Organising Migrating Algorithm

SOMA is not based on the philosophy of evolution (two parents create one new individual -the offspring), but on the behavior of a social group of individuals, e.g. a herd of animals looking for food. One can classify SOMA as an evolutionary algorithm, because the final result, after one migration loop, is equivalent to the result from one generation derived by the classic EA algorithms - individuals hold new positions on the N dimensional hyper-plane. When the group of individuals is created, then the rule mentioned above governs the behavior of all individuals so that they demonstrate 'self-organization' behavior. Because no new individuals are created, and only existing ones are moving over the N-dimensional hyper-plane, this algorithm has been termed the Self-Organizing Migrating Algorithm, or SOMA for short.

SOMA is a stochastic optimization algorithm that is modeled on the social behavior of cooperating individuals, such as swarm algorithms. SOMA works on a population of candidate solutions in loops called *migration loops*. The population is initialized randomly distributed over the search space at the beginning of the search. In each loop, the population is evaluated and the solution with the highest fitness becomes the *Leader L*. Apart from the leader, in one migration loop, all individuals will traverse the input space in the direction of the leader. Mutation, the random perturbation of individuals, is an important operation for evolutionary algorithms. It ensures the diversity amongst the individuals and it also provides the means to restore lost information in a population. Mutation is different in SOMA compared with other EAs. SOMA uses a parameter called PRT to achieve perturbation. This parameter has the same effect for SOMA as mutation has for GA. The novelty of this approach is that the PRT Vector is in canonical version created before an individual starts

its journey over the search space. The PRT Vector defines the final movement of an active individual in search space. The randomly generated binary perturbation vector controls the allowed dimensions for an individual. If an element of the perturbation vector is set to zero, then the individual is not allowed to change its position in the corresponding dimension. An individual will travel a certain distance (called the *path length*) towards the *Leader* in n steps of defined length. If the path length is chosen to be greater than one, then the individual will overshoot the leader. This path is perturbed randomly.

2.1 SOMA Principles and Control Parameters

In the previous sections it was mentioned that SOMA was inspired by the competitive-cooperative behavior of intelligent creatures solving a common problem. Such a behavior can be observed anywhere in the world. A group of animals such as wolves or other predators may be a good example. If they are looking for food, they usually cooperate and compete so that if one member of the group is successful (it has found some food or shelter) then the other animals of the group change their trajectories towards the most successful member. If a member of this group is more successful than the previous best one (is has found more food, etc.) then again all members change their trajectories towards the new successful member. It is repeated until all members meet around one food source. This principle from the real world is of course strongly simplified. Yet even so, it can be said it is that competitive-cooperative behavior of intelligent agents that allows SOMA to carry out very successful searches. For the implementation of this approach, the following analogies are used:

1. Members of herd/pack $\Leftrightarrow$ individuals of population, $PopSize$ parameter of SOMA.
2. Member with the best source of food $\Leftrightarrow$ Leader, the best individual in population for actual migration loop.
3. Food $\Leftrightarrow$ fitness, local or global extreme on N dimensional hyper-plane.
4. Landscape where pack is living $\Leftrightarrow$ N dimensional hyper-plane given by cost function.
5. Migrating of pack members over the landscape $\Leftrightarrow$ migrations in SOMA.

The following section explains in a series of detailed steps how SOMA actually works. SOMA works in loops - so called $Migration$ loops. These play the same role as *Generations* in classic EAs. The difference between SOMA's $Migrationloops$ and EA's *Generations* come from the fact that during a *Generations* in classic EA's offspring is created by means of at least two or more parents (two in GA, four in DE for example). In the case of SOMA, there is no newly created offspring based on parents crossing. Instead, new positions are calculated for the individuals traveling towards the current Leader. The term $Migrations$ refers to their movement over the landscape-hyper-plane. It can be demonstrated that SOMA can be viewed as an algorithm based on offspring creation. The *Leader* plays the role of roe-buck (male), while other individuals play the role of roe (female); note that this has the characteristics of pack reproduction with one dominant male. Hence, GA, DE, etc. may be seen as a special case of SOMA and vice versa (see later SOMA strategy AllToAll). Because

the original idea of SOMA is derived from competitive-cooperative behavior of intelligent beings, we suppose that this background is the most suitable one for its explanation. The basic version of SOMA consists of the following steps:

1. Parameter definition. Before starting the algorithm, SOMA's parameters, e.g. *Specimen*, *Step*, $PathLength$, $PopSize$, PRT, $MinDiv$, $Migrations$ and the cost function needs to be defined. Cost function is simply the function which returns a scalar that can directly serve as a measure of fitness. The cost function is then defined as a model of real world problems, (e.g. behavior of controller, quality of pressure vessel, behavior of reactor, etc.).

2. Creation of Population. A population of individuals is randomly generated. Each parameter for each individual has to be chosen randomly from the given range [Lo, Hi] by using Eq. 1. The population (Fig. 1) then consists of columns - individuals which conform with the specimen.

3. Migrating loop. Each individual is evaluated by cost function and the *Leader* (individual with the highest fitness) is chosen for the current migration loop. Then all other individuals begin to jump, (according to the *Step* definition) towards the *Leader*. Each individual is evaluated after each jump using the cost function. The jumping (Eq. 2) continues, until a new position defined by the $PathLength$ has been reached. The new position after each jump is calculated by Eq. 2. This is shown graphically in Fig. **??**. The individual returns then to that position where it found the best fitness on its trajectory. Before an individual begins jumping towards the *Leader*, a random number is generated (for each individual's component), and then compared with PRT. If the generated random number is larger than PRT, then the associated component of the individual is set to 0 by means of the $PRTVector$ (see Eq. 3 otherwise set to 1. Hence, the individual moves in the N-k dimensional subspace, which is perpendicular to the original space. This fact establishes a higher robustness of the algorithm. Earlier experiments have demonstrated that, without the use of PRT, SOMA tends to determine a local optimum rather than the global one. $Migration$ can be also viewed as a competitive-cooperative phase. During the competitive phase each individual tries to find the best position on its way and also the best from all individuals. Thus during migration, all individuals compete among themselves. When all individuals are in new positions, they release information as to their cost value. This can be regarded as a cooperative phase. All individuals cooperate so that the best individual (*Leader*) is chosen. Competitive-cooperative behavior is one of the other important attributes typical for memetic algorithms.

4. Test for stopping condition. If the difference between *Leader* and the worst individual is not lower than the $MinDiv$ and the maximum number of $Migrations$ has not been reached, return to step 3 otherwise go to step 5 .

5. Stop. Recall the best solution(s) found during the search.

$$InitilPopulation = x_j^{(lo)} + rand_j\,[0,1] \times (x_j^{(hi)} - x_j^{(lo)}) \tag{1}$$

$$x_{i,j}^{ML+1} = x_{i,j,start}^{ML} + (x_{L,j}^{ML} - x_{i,j,start}^{ML}) \cdot t \cdot PRTVector_j \tag{2}$$

$$if\ rnd_j < PRT\ then\ PRTVector_j = 1\ \ else\ 0\ , \quad j = 1, \ldots, N \tag{3}$$

Steps 1 to 5 are graphically depicted in Figure 1 or in pseudocode in Figure 2.

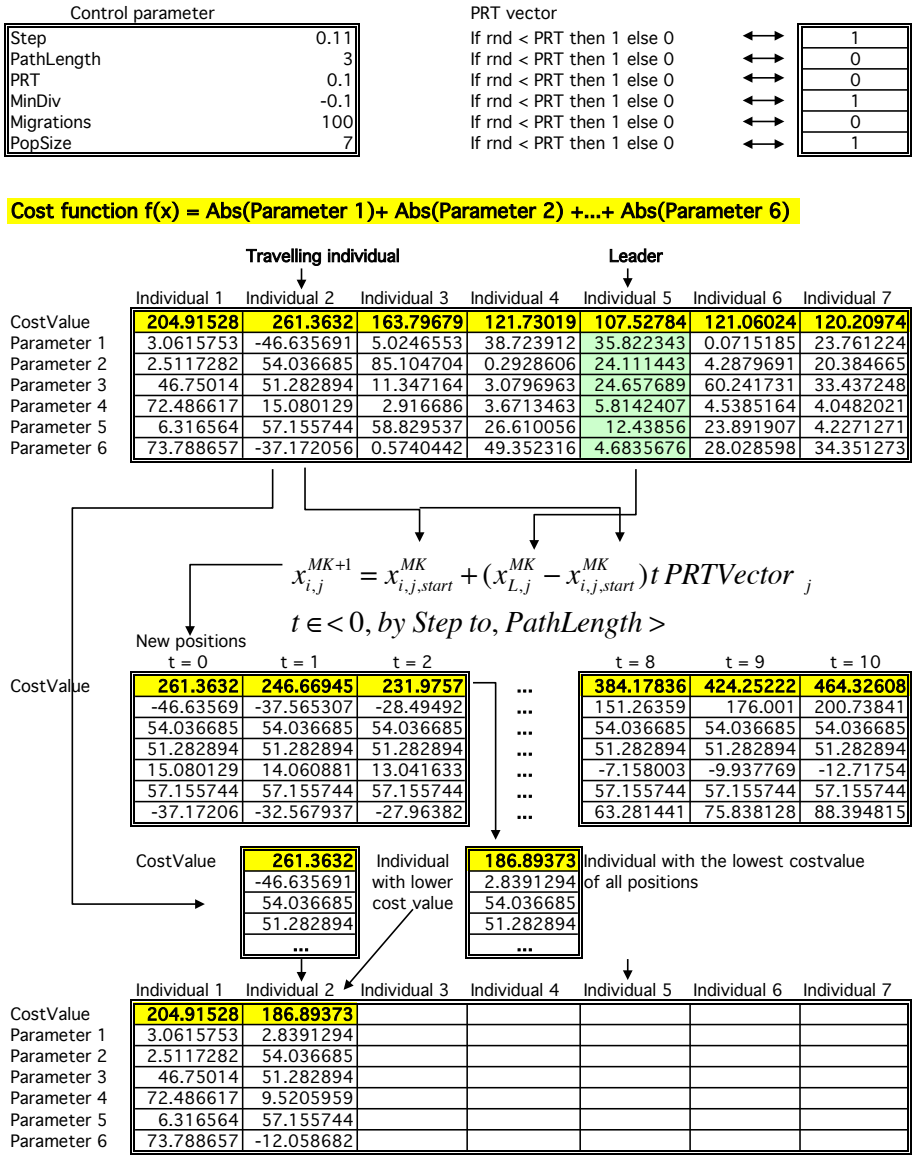

Figure 1: SOMA Principle

The pseudocode of SOMA can be written such as:

SOMA AllToOne input parameters :
x : *the initial randomly generated population*
Controlling and stopping parameters − see Tab. 1
$\mathbf{f_{cost}}$: *cost function (fitness function)*
Specimen : *an individual structure* (*parameters range, its "nature" i.e. real, integer, discrete, ...*)
  for $i \leq Migration$ **do**
    begin
    Selection of the best individual − Leader
      for$j \leq PopSize$ **do**
        selection of j_{th} *individual*
        calculate f_{cost} *of the new positions see Eq.*2
        save the best solution of the j_{th} *individual on its trajectory in a new population*
      end
    if $MinDiv < |\text{best\_individual} - \text{worst\_individual}|$
      then begin
        Stop SOMA and return the best solution (*or last calculated population*)
      end
    end
Stop SOMA and return the best solution (*or last population*)

---

or see Eq. 2. SOMA principle can be graphically visualized as in Fig. 1.

Input :$N, Migrations(ML), PopSize \geq 2, PRT \in [0,1], Step \in (0,1]$, MinDiv $\in (-\infty, \infty)$, PathLength $\in (1,5]$, *Specimen* with upper and lower bound $x_j^{(hi)}, x_j^{(lo)}$

$$\text{Inicialization} : \begin{cases} \forall i \leq PopSize \wedge \forall j \leq N : x_{i,j}^{ML_0} = x_j^{(lo)} + rand_j\,[0,1]\left(x_j^{(hi)} - x_j^{(lo)}\right) \\ i = \{1,2,...,Migrations\}, j = \{1,2,...,N\} \end{cases}$$

$$\begin{cases} \text{While } i < Migrations \\ \forall i \leq PopSize \begin{cases} \text{While } t \leq PathLength \\ if\ rnd_j < PRT\ then\ PRTVector_j = 1\ \ else\ 0\ , \quad j = 1,\ldots,N \\ x_{i,j}^{ML+1} = x_{i,j,start}^{ML} + (x_{L,j}^{ML} - x_{i,j,start}^{ML})\,t\,PRTVector_j \\ f\left(x_{i,j}^{ML+1}\right) = \text{ if } f\left(x_{i,j}^{ML}\right) \leq f\left(x_{i,j,start}^{ML}\right) \text{ else } f\left(x_{i,j,start}^{ML}\right) \\ t = t + Step \end{cases} \\ i = i + 1 \end{cases}$$

Figure 2: Self Organizing Migrating Algorithm

Based on above described principles, SOMA can be also regarded as a member of swarm intelligence class algorithms. In the same class is the algorithm particle swarm, which is also based on population of particles, which are mutually influenced amongst themselves.

2.2 SOMA Strategies

Currently, a few variations - strategies of the SOMA algorithm exist. All versions are almost fully comparable with each other in the sense of finding of global optimum. These versions are:

1. 'AllToOne': This is the basic strategy, that was previously described. Strategy AllToOne means that all individuals move towards the Leader, except the Leader. The *Leader* remains at its position during a $Migration$ loop. The principle of this strategy is shown in Figure 3.

2. 'AllToAll': In this strategy, there is no Leader. All individuals move towards the other individuals. This strategy is computationally more demanding. Interestingly, this strategy often needs less cost function evaluations to reach the global optimum than the AllToOne strategy. This is caused by the fact that each individual visits a larger number of parts on the N dimensional hyper-plane during one $Migration$ loop than the AllToOne strategy does. Figure 4 shows the AllToAll strategy with $PRT = 1$.

3. 'AllToAll Adaptive': The difference between this and the previous version is, that individuals do not begin a new migration from the same old position (as in AllToAll), but from the last best position found during the last traveling to the previous individual.

4. 'AllToRand': This is a strategy, where all individuals move towards a randomly selected individual during the migration loop, no matter what cost value this individual has. It is up to the user to decide how many randomly selected individuals there should be. Here are two sub-strategies:

   - The number of randomly selected individuals is constant during the whole SOMA process.
   - For each migration loop, (in intervals of [1, $PopSize$]) the actual number of individuals is determined randomly. Thus, the number of randomly chosen individuals in the second sub-strategy is different in each migration loop.

2.3 SOMA Parameters

SOMA, as other EAs, is controlled by a special set of parameters. Some of these parameters are used to stop the search process when one of two criteria are fulfilled; the others are responsible for the quality of the results of the optimization process. The parameters are shown in Table 1

A sensitivity of SOMA, as well as of other EAs, is that it has a slight dependence on the control parameter setting. During various tests it was found that SOMA is sensitive on the parameter setting as well as others algorithms. On the other side there was found setting that is almost universal, i.e. this setting was used almost in all simulations and experiments with very good performance of SOMA. The control parameters are described

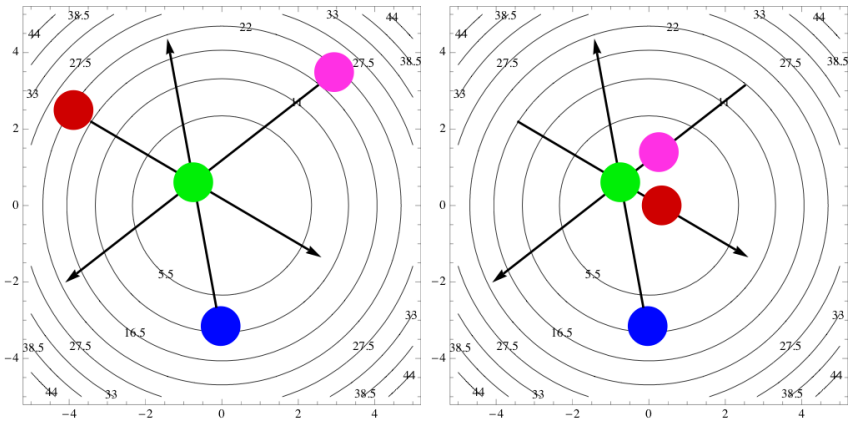

Figure 3: SOMA AllToOne, the principle of migrating (left) and new individual position (right) after one migration

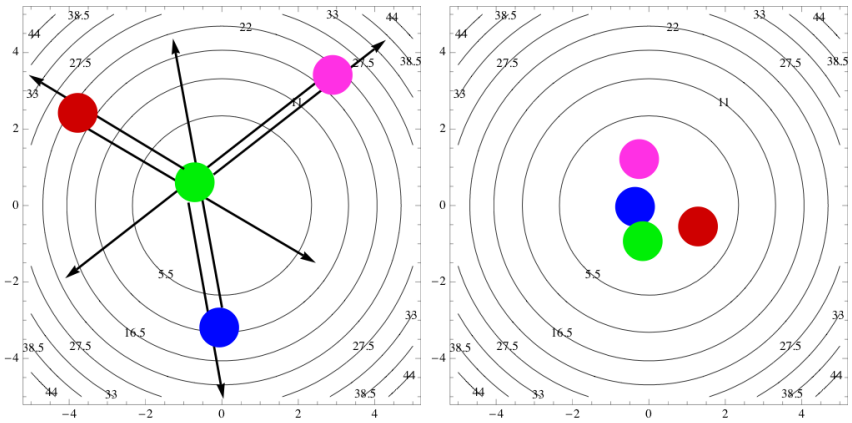

Figure 4: SOMA AllToAll, the principle of migrating (left) and new individual position (right) after one migration

below and recommended values for the parameters, derived empirically from a great number of experiments, are given:

- *PathLength* $\in$ [1.1, 5]. This parameter defines how far an individual stops behind the *Leader* (*PathLength*=1: stop at the leader's position, *PathLength*=2: stop behind the leader's position on the opposite side but at the same distance as at the starting point). If it is smaller than 1, then the Leader's position is not overshot, which carries the risk of premature convergence. In that case SOMA may get trapped in a local optimum rather than finding the global optimum. Recommended value is **3–5.**

- *Step* $\in$ [0.11, *PathLength*]. The step size defines the granularity with what the search space is sampled. In case of simple objective functions (convex, one or a few local extremes, etc.), it is possible to use a large *Step* size in order to speed up the search process. If prior information about the objective function is not known, then the recommended value should be used. For greater diversity of the population, it is better if the distance between the start position of an individual and the *Leader* is not an

Table 1: SOMA parameters

Parameter name	**Recommended range**	**Remark**
PathLength	[1.1, 5]	Controlling parameter
Step	[0.11, *PathLength*]	Controlling parameter
PRT	[0, 1]	Controlling parameter
Dim	Defined by problem	Number of arguments in cost function
PopSize	[10, up to user]	Controlling parameter
Migrations	[10, up to user]	Stopping parameter
MinDiv	[arbitrary negative, up to user]	Stopping parameter

integer multiple of the *Step* parameter. That means that a *Step* size of 0.11 is better than a *Step* size of 0.1 (that lead jumping directly on the *Leader* position), because the active individual will not reach exactly the position of the *Leader*. Recommended value is **0.11.**

- $PRT \in [0, 1]$. *PRT* stands for perturbation. This parameter determines whether an individual will travel directly towards the *Leader*, or not. It is one of the most sensitive control parameters. The optimal value is near 0.1. When the value for *PRT* is increased, the convergence speed of SOMA increases as well. In the case of low dimensional functions and a great number of individuals, it is possible to set *PRT* to 0.7-1.0. If *PRT* equals 1 then the stochastic component of SOMA **disappears** and it performs only deterministic behavior suitable for local search.

- *Dim* - the dimensionality (number of optimized arguments of cost function) is given by the optimization problem. Its exact value is determined by the cost function and usually cannot be changed unless the user can reformulate the optimization problem. Recommended value is $\mathbf{0.1 - 0.2.}$

- $PopSize \in [10,$ up to the user]. This is the number of individuals in the population. It may be chosen to be 0.5 to 0.7 times of the dimensionality (*Dim*) of the given problem. For example, if the optimization function has 100 arguments, then the population should contain approximately 30-50 individuals. In the case of simple functions, a small number of individuals may be sufficient; otherwise larger values for *PopSize* should be chosen. It is recommended to use at least 10 individuals (two are theoretical minimum), because if the population size is smaller than that, SOMA will strongly degrade its performance to the level of simple and classic optimization methods. Recommended value is $\mathbf{10 >.}$

- $Migrations \in [10,$ up to user]. This parameter represents the maximum number of iterations. It is basically the same as generations for GA or DE. Here, it is called *Migrations* to refer to the nature of SOMA - individual creatures move over the landscape and search for an optimum solution. *Migrations* is a stopping criterion, i.e. it tells the optimizing process when to stop. Recommended value is up to user

experience, generally **10 >.**

- $MinDiv \in$ [arbitrary negative (switch off this criterion), up to the user]. The $MinDiv$ defines the largest allowed difference between the best and the worst individual from actual population. If the difference is smaller then defined $MinDiv$, the optimizing process is will stop (see Fig. 5). It is recommended to use small values. It is safe to use small values for the $MinDiv$, e.g. $MinDiv = 1$. In the worst case, the search will stop when the maximum number of migrations is reached. Negative values **are also possible** for the $MinDiv$. In this case, the stop condition for $MinDiv$ will not be satisfied and thus SOMA will pass through all $Migrations$.

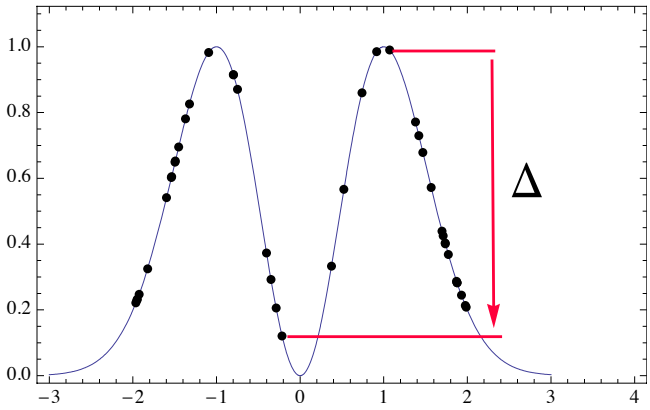

Figure 5: $MinDiv$ principle

3 Firefly Algorithm

Firefly algorithm (FA) is a new swarm intelligence algorithm developed by Yang in 2010. It is inspired by the social behavior of fireflies based on the flashing and attraction characteristics of fireflies. In the past five years, the research of FA has attracted much attention. Different versions of FA has been designed to solve bench- mark or real-world optimization problem

In the FA, the fitness function for a given problem is associated with the light intensity. The brighter the firefly is, the better the firefly is. That means a brighter firefly has a better fitness value. The search process of FA depends on the attractions between fireflies. Based on these attractions, a firefly tends to move other brighter fireflies. If a firefly is brighter than another one, the brighter firefly will not conduct any search. When the current firefly is brighter than another one, a local search operation is conducted on the current one to provide more chances of finding more accurate solutions.

3.1 Description

As mentioned before, the FA mimics the behavior of the social behavior of the flashing characteristics of fireflies. To simply the behavior of fireflies and construct the search mode of FA, three rules are used as follows:

1. All fireflies are unisex so that one firefly is attracted to other fireflies regardless of their sex;

2. Attractiveness is proportional to their brightness. For any two fireflies, the less bright one is attracted by the brighter one. The attractiveness is proportional to the brightness and they both decrease as their distance increases. If no one is brighter than a particular firefly, it moves randomly;

3. The brightness or light intensity of a firefly is affected or determined by the land- scape of the objective function to be optimized. For a minimization problem, the brightness can be proportional to the objective function. It means that the brighter firefly has smaller objective function value.

As light intensity and thus attractiveness decreases as the distance from the source increases, the variations of light intensity and attractiveness should be monotonically decreasing functions. This can be approximated by the following Equation (4).

$$I\left(r\right) = I_0 e^{-\gamma r^2} \tag{4}$$

where I is the light intensity, I_0 is the original light intensity, and c is the light absorption coefficient. The attractiveness of a firefly is proportional to the light intensity. The attractiveness β of a firefly can be defined by Equation (5):

$$\beta\left(r\right) = \beta_0 e^{-\gamma r^2} \tag{5}$$

where β_0 is a constant and presents the attractiveness at $r = 0$. The distance between $r_{i,j}$ between any two fireflies i and j can be calculated by Equation (6):

$$r_{i,j} = \left\| X_i - X_j \right\| = \sqrt{\sum_{d=1}^{D} \left(x_{i,d} - x_{j,d}\right)^2} \tag{6}$$

where D is the dimensional size of the given problem. Based on the above definitions, the movement of this attraction is defined by Equation (7):

$$x_{i,d}^{(t+1)} = x_{i,d}^{(t)} + \beta_0 \cdot e^{-\gamma r_{i,j}^2} \cdot \left(x_{j,d}^{(t)} - x_{i,d}^{(t)}\right) + \alpha \cdot \varepsilon_{i,d}^{(t)} \tag{7}$$

where $x_{i,d}$ and $x_{j,d}$ is the d^{th} dimension of firefly i and j, respectively, a is a random value with the range of [0,1], $\varepsilon_{i,d}$ is a Gaussian random number for the d^{th} dimension, and t indicates the index of generation.

The operating parameters for the FA algorithm is given in Table 2.

The general outline of the Firefly algorithm is given in the following Algorithm 1.

Table 2: Table of FA parameters

Parameter	value
α	0.5
β_0	0.2
γ	1.0

input : *Iterations*: maximum number of iterations
D: dimension of the problem
Fireflies: number of fireflies f_i
I: light intensity
γ: light absorption coefficient
Bounds: Problem bounds (*U* - upper bound, *L* - lower bound)
output: *gBest*: best firefly found

```
for i = 1,...,Fireflies do
    /* generate fireflies randomly                                    */
    f_i = L+rand[0,1](U-L)
    /* calculate particles fitness                                    */
    I_i = f(f_i)
end for
for t = 1,...,Iterations do
    for i = 1,...,Fireflies do
        for j = 1,...,Fireflies do
            if I_j < I_i then
                /* Move firefly j towards firefly i (Eqn.6)           */
                /* Attractiveness varies with distance r via exp[-γr]
                   (Eqn.5)                                            */
                /* evaluate and update the worst firefly in population
                   (Eqn.7)                                            */
            end if
        end for
    end for
end for
```

Algorithm 1: Firefly Algorithm

4 Harmony Search

In order to explain the Harmony Search in more detail, let us first idealize the improvisation process by a skilled musician. When a musician is improvising, he or she has three possible choices:

- play any famous piece of music (a series of pitches in harmony) exactly from his or her memory;

- play something similar to a known piece (thus adjusting the pitch slightly);
- compose new or random notes. Zong Woo Geem et al. formalized these three options into quantitative optimization process in 2001, and the three corresponding components become: usage of harmony memory, pitch adjusting, and randomization

The usage of harmony memory is important, as it is similar to the choice of the best-fit individuals in genetic algorithms. This will ensure that the best harmonies will be carried over to the new harmony memory. In order to use this memory more effectively, it is typically assigned as a parameter $r_{accept} \in [0, 1]$, called harmony memory accepting or considering rate. If this rate is too low, only few best harmonies are selected and it may converge too slowly. If this rate is extremely high (near 1), almost all the harmonies are used in the harmony memory, then other harmonies are not explored well, leading to potentially wrong solutions. Therefore, typically, we use $r_{accept} = 0.7$ to 0.95.

The second component is the pitch adjustment determined by a pitch bandwidth b_{range} and a pitch adjusting rate r_{pa}. Though in music, pitch adjustment means to change the frequencies, it corresponds to generate a slightly different solution in the Harmony Search algorithm. In theory, the pitch can be adjusted linearly or nonlinearly, but in practice, linear adjustment is used. So we have

$$x_{new} = x_{old} + b_{range} \cdot \varepsilon \tag{8}$$

where x_{old} is the existing pitch or solution from the harmony memory, and x_{new} is the new pitch after the pitch adjusting action. This essentially produces a new solution around the existing quality solution by varying the pitch slightly by a small random amount $[1, 2]$. Here ε is a random number generator in the range of $[-1, 1]$. Pitch adjustment is similar to the mutation operator in genetic algorithms. We can assign a pitch-adjusting rate (r_{pa}) to control the degree of the adjustment. A low pitch adjusting rate with a narrow bandwidth can slow down the convergence of HS because the limitation in the exploration of only a small subspace of the whole search space. On the other hand, a very high pitch-adjusting rate with a wide bandwidth may cause the solution to scatter around some potential optima as in a random search. Thus, we usually use $r_{pa} = 0.1$ to 0.5 in most applications.

The third component is the randomization, which is to increase the diversity of the solutions. Although adjusting pitch has a similar role, but it is limited to certain local pitch adjustment and thus corresponds to a local search. The use of randomization can drive the system further to explore various diverse solutions so as to find the global optimality.

```
Function Harmony Search() /* Harmony Search Algorithm routines */
    Objective function f(x), x = (x_1, x_2, ..., x_d)^T
    Generate initial harmonics (real number arrays)
    Define pitch adjusting rate (r_pa), pitch limits and bandwidth
    Define harmony memory accepting rate (r_accept)
    while !Terination_Criteria do
        Generate new harmonics by accepting best harmonics
        Adjust pitch to get new harmonics (solutions)
        if rand > r_accept then
            choose an existing harmonic randomly;
        else if rand > r_pa then
            adjust the pitch randomly within limits;
        else generate new harmonics via randomization;
        Accept the new harmonics (solutions) if better
    end while
    Find the current best solution
```

Algorithm 2: Harmony Search Algorithm

The three components in harmony search can be summarized as the pseudocode shown in Algorithm 2. In this pseudocode, we can see that the probability of randomization is:

$$P_{random} = 1 - r_{accept} \tag{9}$$

and the actual probability of adjusting pitches is

$$P_{pitch} = r_{accept} \cdot r_{pa} \tag{10}$$

5 Experimentation

The student is required to code all three algorithms in the language of their choice. Ideally, all three algorithms should share same auxiliary structures, such as population generation, memory management, problems definitions etc.

The experimentation parameters is given in Table 3.

Table 3: Experiment parameters

Parameters	Values
Population size	50 (min)
Iterations	100 (min)
Dimensions	20

Submission

The student must submit the following separate files to canvas:

1. source codes for the problems

2. a LaTeX typeset report on the results and its analysis

The report must contain an introduction in the algorithms, the full experimentation results in tabular format and condensed results with statistical analysis compared with what was obtained in Lab 3.

The files must be submitted through Canvas by midnight November 21, 2016. The penalty for late submission is 10% for 1 day, 20% for 2 day, after which it will be zero. The grading rubric is given in Table 4.

Table 4: Grading rubric

File	Aspects	Points
`Code`	Compiles and executes	35
	Explanation	15
`Report`	Results	25
	Analysis	25