# 20

# Practical Considerations for Simulated Annealing Implementation

Sergio Ledesma, Gabriel Aviña and Raul Sanchez
*School of Engineering - University of Guanajuato, Mexico*

## 1. Introduction

Nowadays, there are many optimization problems where exact methods do not exist or where deterministic methods are computationally too complex to implement. Simulated annealing may be the answer for these cases. It is not greedy in the sense that it is not fool with false minima, and is pretty easy to implement. Furthermore, because it does not require a mathematical model, it can be used to solve a broad range of problems.

Unfortunately, mapping a real problem to the domain of simulated annealing can be difficult and requires familiarity with the algorithm. More often than not, it is possible to encode the solution (solve the problem) using several approaches. In addition, there are other factors that determine the success of failure of this algorithm. This chapter reviews how to plan the encoding of the solution, and discusses how to decide which encoding is more appropriate for each application.

Several practical considerations for the proper implementation of simulated annealing are reviewed and analyzed. These include how to perturb the solution, how to decide a proper cooling schedule, and most important, how to properly implement the algorithm. Several cooling schedules are covered, including exponential, linear and temperature cycling. Additionally, the impact of random number generators is examined; how they affect the speed and quality of the algorithm. Essentially, this chapter is focused for those who want to solve real problems using simulated annealing for artificial intelligence, engineering, or research.

An illustrative example is solved using simulated annealing and implemented in a popular programming language using an object-oriented approach. This chapter offers a great opportunity to understand the power of this algorithm as well as to appreciate its limitations.

Finally, it is reviewed how is possible to combine simulated annealing with other optimization algorithms (including the deterministic ones) to solve complex optimization problems. In particular, it is discussed how to train artificial neural networks using simulated annealing with gradient based algorithms.



Source: Simulated Annealing, Book edited by: Cher Ming Tan, ISBN 978-953-7619-07-7, pp. 420, February 2008, I-Tech Education and Publishing, Vienna, Austria

## 2. Simulated annealing basics

Simulated annealing is an optimization method that imitates the annealing process used in metallurgic. Generally, when a substance goes through the process of annealing, it is first heated until it reaches its fusion point to liquefy it, and then slowly cooled down in a control manner until it solids back. The final properties of this substance depend strongly on the cooling schedule applied; if it cools down quickly the resulting substance will be easily broken due to an imperfect structure, if it cools down slowly the resulting structure will be well organized and strong.

When solving an optimization problem using simulated annealing the structure of the substance represents a codified solution of the problem, and the temperature is used to determined how and when new solutions are perturbed and accepted. The algorithm is basically a three steps process: perturb the solution, evaluate the quality of the solution, and accept the solution if it is better than the new one.

To implement simulated annealing, it is usually necessary to generate huge amounts of random numbers. Unfortunately, typical random generators included in programming languages are of low quality, and are not useful for simulated annealing. These random sequences have a finite length and may have correlation. Choosing an appropriate random generator requires specific knowledge of the problem; basically it is important to establish the amount of random numbers that will be required and the speed of the generator (some problems may require quality or speed; some others will require both quality and speed). (Press et al., 2002) provides a comprehensive review on the subject and includes actual code to implement high quality random number generators.

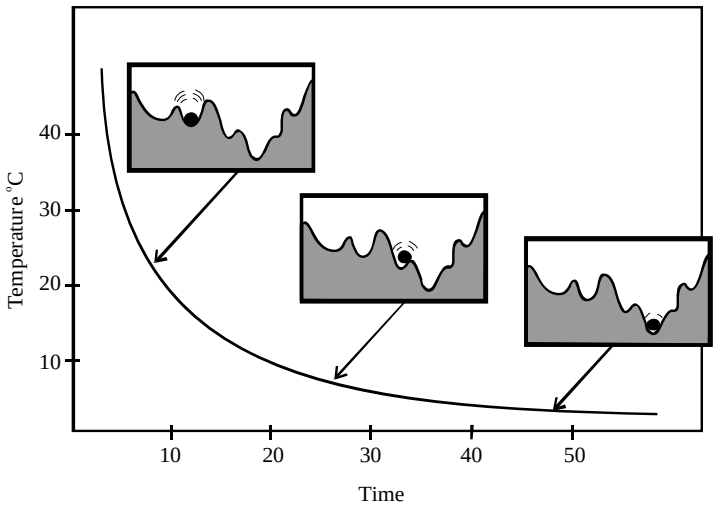


Fig. 1. The method of simulated annealing.

The method of simulated annealing can be easily understood by observing Fig. 1 which shows a hermetic box with an internal uneven surface (with peaks and valleys), and a ball resting on this surface. The objective is to move the ball to a position as close as possible to the bottom of the box. At the beginning of the process the temperature is high and strong perturbations are applied to the box allowing the ball to easily jump over high peaks in

search of the bottom of the box. Because of the high energy applied, it is possible for the ball to go down or up easily. As time goes by, the temperature decreases and the ball has less energy to jump high peaks. When the temperature has decreased to the point when the ball is able to jump only very small peaks, the ball should be hopefully very close to the bottom of the box and the process is completed. As it can be from this example, many things can go wrong with simulated annealing; these problems and how to avoid them will be discussed in Section 4.

## 3. Advantages of using simulated annealing.

### 3.1 A mathematical model is not required

As odd at it sounds, some problems in real life do not have an exact model, or sometimes, the model is too complicated to be useful. In other cases, there is so little information about the problem that existing models cannot be appropriately used. For these cases, simulated annealing may be perfect as long as two basic operations can be implemented: perturb and evaluate. Thus, if a solution can be designed so that it can be perturbed and evaluated, then the problem can be solved using simulated annealing for sure.

### 3.2 The problem has many solutions and some of them are not optimal

Unfortunately, some problems have their solution surrounded by non optimal solutions (false minima), and typical optimization algorithms may have a bad time trying to escape from these false solutions. Consider a problem that could be described as a system of non-linear equations, clearly, the mean-squared error may be used to measure the quality of the solution.

For these problems, the mean-squared error is compute using the actual output of the system and the desired output of it. Generally, gradient based algorithms are a good choice to minimize the mean-squared error and find a solution, as they required much less time than solving the problem if simulated annealing is used. However, gradient based algorithms require knowledge of the derivative of the error with respect to each unknown and they are useful when the global minimum is clearly defined. On the other hand, simulated annealing does not required derivative information, and it is not easily fooled with local minima.

Before moving our attention to another topic, it is important to mention that simulated annealing and gradient based algorithms can be used together as hybrids algorithms for global optimization. First, simulated annealing is used to find a rough estimate of the solution, then, gradient based algorithms are used to refine the solution (Masters, 1993); note that more research is needed to optimize and blend simulated annealing with other optimization algorithms and produce hybrids.

## 4. Typical problems when using simulated annealing.

### 4.1 Initial temperature is too high

Consider again Fig. 1 that described pictorially the process of simulated annealing, at high temperatures the ball has enough energy to jump over high peaks and it can go easily up or down. If the initial temperature is too high, the ball may fall down and reach a position close to the bottom, but it is also very likely that the ball may jump up ending in a position even higher than the initial position. In other words, applying too much perturbation is useless

and should be avoided. This raises the questions: how much perturbation should be applied? How long a level or perturbation should be applied?

### 4.2 Temperature goes down to quickly

As it can be seen from the previous example, at high temperatures the method of simulated annealing is searching for the global minimum in a broad region, and as the temperatures decreases the method is reducing this search region and tries mainly to refine the solution found at high temperatures. This is one of the good qualities that makes simulated annealing superior when the problem at hand has several deep valleys. Simulated annealing does not easily fall down into a deep valley located close by, instead it searches in an ample area trying always to go down and very occasionally up as the temperature allows. On the other hand, typical optimization methods fall quickly into a close deep valley even if it not the deepest valley. Thus, it is important to note that the temperature must go down slowly allowing the method to search thoroughly at each temperature.

There are two typical cooling schedules in the literature: exponential and linear. Fig. 1 shows a typical exponential cooling, as it can be seen from this figure, the process spends little time at high temperatures, and as the temperature decreases more and more time is spend at each temperature, allowing the algorithm to refine very well the solution found at high temperatures. On linear cooling, the temperature decreases linearly as the time increases, thus, the algorithm spends the same amount of time at each temperature. Clearly, linear cooling must be used when there are several deep valleys close by (note the quality of the final solution using linear cooling may not be good). On temperature cycling the temperature goes down and up cyclically refining the quality of the solution at each cycle. As it was indicated in (Ledesma et al., 2007), temperature cycling is beneficial for training of auto associative neural networks. Additionally, it has been pointed out (Reed & Marks, 1999) that a temperature reduction schedule inversely proportional to the logarithm of time will guarantee converge (in probability) to a global minimum, however, in practice this schedule takes too long, and it is often more efficient to repeat the algorithm a number of times using a faster schedule. Other cooling schedules area described in (Luke, 2007).

### 4.3 Process completes and the solution is not optimal

At the beginning of the process, the temperature and error are high, as time goes by, the temperature decreases slowly spending some time at each temperature and the error should be hopefully also decreasing. However, it is possible that the process completes without finding an optimal solution. Carefully selecting the parameters of simulated annealing may reduce the probability of this to happen, but this can happen. An easy solution to increase the probability of success is to try again and again until a desired error is obtained.

## 5. Simulated annealing implementation

Simulated annealing is a two steps process: perturb, and then evaluate the quality of the solution. Usually, the algorithm uses the solution error to make decisions about the acceptance of a new solution. Next, some notation will be offered to make a clear presentation. Let represent a problem solution of M variables as

$$\mathbf{X} = \{x_1, x_2, x_3, \dots, x_M\} \tag{1}$$

where $x_1, x_2, x_3, \ldots, x_M$ are to be found by means of simulated annealing. This representation may be useful for most optimization algorithms; however, simulated annealing is a temperature dependent algorithm and the process temperature must be introduced in Equation 1. Let define T as the process temperature

$$T = T_1, T_2, T_3, \ldots, T_N \tag{2}$$

where $T_1$ is the initial temperature, $T_N$ is the final temperature, N is the number of temperatures, and the values of T are chosen following a specific cooling schedule that is problem dependent. Please note that T has been defined as a discrete variable because usually the temperature does not increase continually.

To improve the performance of the method of simulated annealing, it is usual to spend some time at each temperature by performing a fixed number of iterations before decreasing the temperature. Let K be the number of iterations performed at each temperature, then Equation (1) can be written as

$$\mathbf{X}_i = \{x_{1,i}, x_{2,i}, x_{3,i}, \ldots, x_{M,i}\}, \quad i = 1, 2, 3, \ldots \tag{3}$$

where i is the number of perturbations applied to the solution, and $x_{1,i}$ is the value of $x_1$ after it is has been perturbed i-times. Thus, at the end of temperature $T_1$, the number of perturbations applied to the solution is K, and $\mathbf{X}_K$ represents the solution at the end of this temperature. Usually, each solution $\mathbf{X}_i$ must have an error associate with it, let

$$E_1, E_2, E_3, \ldots \tag{4}$$

be the errors of $\mathbf{X}_1, \mathbf{X}_2, \mathbf{X}_3, \ldots$ respectively. Generally, a technique to estimate the solution error must be defined, but typically this technique is problem dependent and full knowledge of the problem is required. Consider for example a problem where five pieces are to be located at discrete positions in the plane x-y, and it is desired that each piece meets some constraints; without a doubt, the error may be defined as the number of pieces that do not meet the constraints. For other optimization problems, the mean squared error may be more appropriate; common sense is required as rigid rules do not exists.

As it can be induced from the previous discussion, there are not regulations that dictate or limit simulated annealing implementation. There are, however, some specific criteria about how to accept a solution once it has been perturbed. One obvious criterion is to accept a solution whenever it has a less error than the previous solution. There is, though, one popular algorithm used to manage simulated annealing. The metropolis algorithm, shown in Equation 5, follows the criterion discussed previously, and is typically used in simulated annealing to compute the probability of acceptance for a perturbed solution.

$$p_a = \begin{cases} e^{-\frac{k\,\Delta E}{T}} & \Delta E > 0 \\ 1 & \Delta E \leq 0 \end{cases} \tag{5}$$

where $\Delta E$ is the difference between the solution error after it has perturbed, and the solution error before it was perturbed, T is the current temperature and k is a suitable constant. A plot of Equation (5) is presented in Figure 2; it can be observed that when $\Delta E$ is

negative the solution is always accepted. However, the algorithm may accept a new solution even if the solution has not a smaller error than the previous one (a positive ΔE), and the probability to do this decreases when the temperature decreases or when ΔE increases. Consequently, at high temperatures the algorithm may wander wildly accepting bad solutions; as the temperature decreases, the algorithm is more selective and accepts perturbed solutions only when the respective ΔE is small. This is the theory behind simulated annealing and should be clearly understood to properly implement the algorithm.

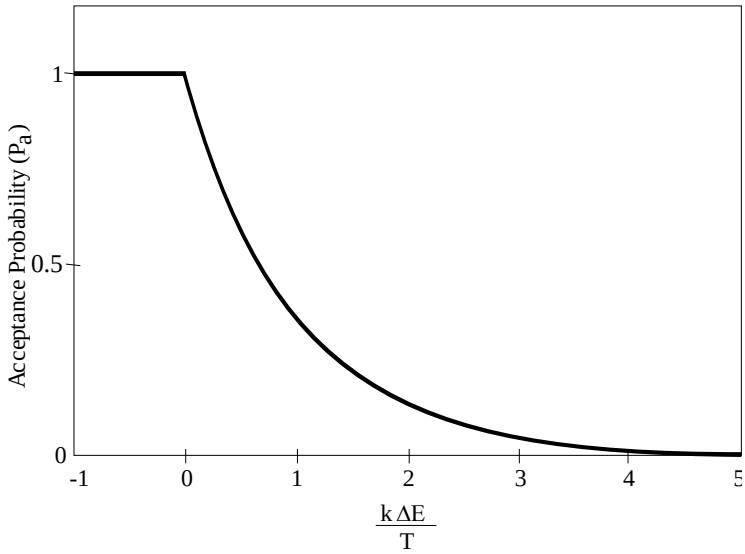


Fig. 2. Probability of acceptance following the Metropolis algorithm

Consider now Figure 3 which shows the probability of acceptance as a function of ΔE for several values of k/T. From this figure, it can be seen that the constant k plays an important role on the algorithm success; if k is equal to T, the algorithm will accept solutions with high probability even if ΔE is not small. This is not good as the method will spend great time trying with bad solutions; even if an excellent solution is found, the method will easily discard it. Generally, a medium ration k/T is desired at the beginning of the process. The authors suggest estimating the value of k as a previous step of the annealing process. This can save a lot of time, as there is not unique value of k that can be used for all optimization problems.

## 6. Estimating k

When an optimization problem is not properly solved using simulated annealing, it may sound suitable to increase the number of temperatures and the number of iterations at each temperature. Additionally, it may sound logical to start at a high temperature and end with a very low final temperature. However, it is most recommended to carefully choose the simulated annealing parameters in order to minimize the number of calculations and, consequently, reduce the time spend on vain perturbations.

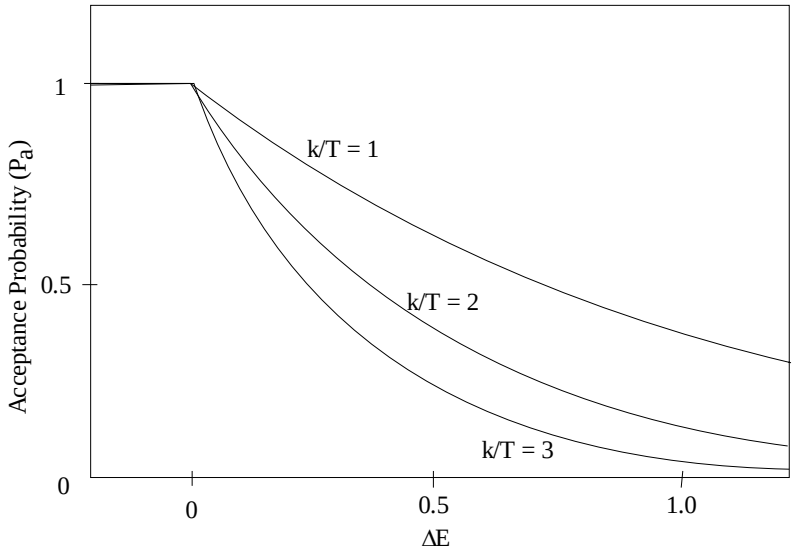


Fig. 3. Probability of acceptance for several values of k / T.

At the beginning of annealing, it is necessary to have an initial solution to start working with. Let $\mathbf{X}_0$ be the initial solution before applying any perturbation to it, and $E_0$ the error associated with $\mathbf{X}_0$. Typically, $\mathbf{X}_0$ may be created by assigning random values to $\{x_1, x_2, x_3, \ldots, x_M\}$, however, in most cases, it is strongly recommended to use the problem requirements to create $\mathbf{X}_0$., this will warranty at least a good starting point.

As it was described before, the constant k plays an important role on simulated annealing for global optimization. In this section, the authors suggest a simple method to estimate k using the essential operations of simulated annealing (perturb and evaluate). After inspecting Equation 9, it is clear that the value of k must be estimated using the initial temperate and the delta error. An estimate for ΔE can be computed from

$$\Delta E = \sigma_E \tag{6}$$

which can be estimated as

$$\Delta E \approx \frac{1}{Q-1}\sum_{i=1}^{Q} E_i - \frac{1}{Q(Q-1)}\sum_{i=1}^{Q}(E_i)^2 \tag{7}$$

that is, the sample variance of E when the solution $\mathbf{X}_0$ has been perturbed Q times. In practice, Equation 7 is an excellent estimator of the initial value of ΔE as long as Q is at least 1000 or more. It is important to mention that an exact value of ΔE is not required as this value is used only to a get rough estimate of k; this implies that a big value for Q is not necessary.

Once an estimate for the delta error of the solution has been found, finding an estimate for k is straightforward as Equation 5 can be directly used to solve for k. However, an initial value for the probability of acceptance needs to be defined. It is clear that the initial probability of acceptance must not be close to one, neither must be close to zero. A value

between 0.7 and 0.9 is recommended. A probability of acceptance bigger than 0.9 has not practical purpose as the algorithm will accept too many bad solutions. On the other hand, a value that is less than 0.7 will rob the algorithm the opportunity to search abroad, loosing one of the main advantages of simulated annealing. In general, an initial value for the probability of acceptance should be 0.8. Thus, an estimate of k can be express as

$$k = -\frac{T_0 \ln(0.8)}{\overline{\sigma}_E} \tag{8}$$

where an estimate for the standard deviation of the solution error can be computed using Equation 7. The performance of the algorithm is dramatically increased when Equation 8 is used because unnecessary and vain perturbations are not computed; instead the algorithm uses this precious CPU time on doing actual work.

## 7. Implementing simulated annealing

For now, the reader should have a sense of how simulated annealing works. However, the reader may have some doubts on how to implement it. As it was established before, common sense is required for proper implementation of the algorithm as there are not hard rules. This section describes how to use a programming language to correctly implement simulated annealing. Figure 4 shows the UML diagram for a class to implement simulated annealing. At the top of the diagram the class name (SimulatedAnnealing) is shown, the second block contains the member variables and the third block the member functions. The member variables' names are self explanatory. However, note that k and finalTemp are declared as private as the class itself will compute these values from the other setup parameters. The only public function is Start, it should be called once we are ready to start the annealing process.

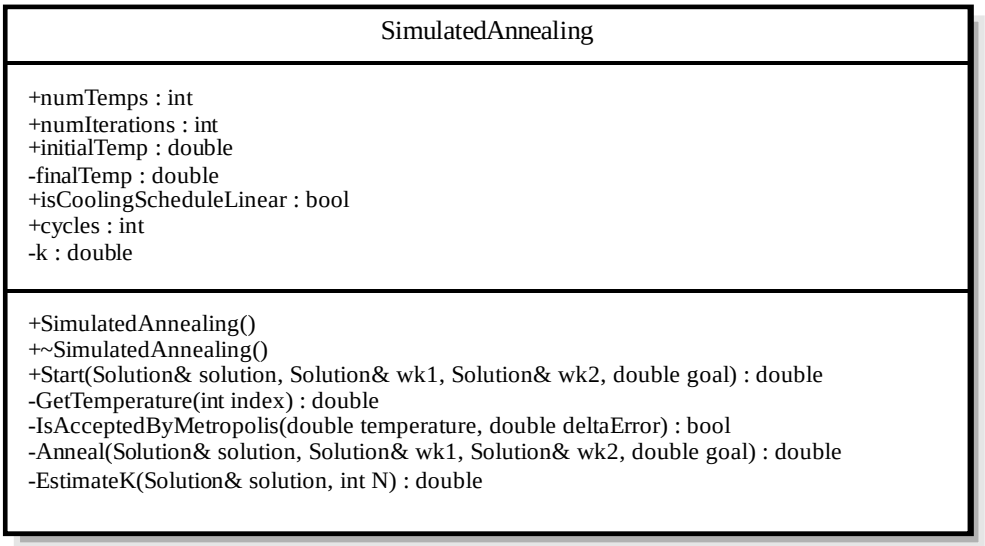


Fig. 4. UML diagram for a class to implement simulated annealing.

The class of Figure 4 makes reference to the abstract class Solution depicted in Figure 5. The class Solution contains the actual implementation of the problem that maps the real problem to the solution coding. Figure 5 describes two classes: Solution at the top and NumEq at the bottom. The class NumEq will be discussed on the next section, for now, just note that NumEq implements the pure abstract functions of the class Solution: operator=, OnInitialize, OnPerturb and OnComputeError. These are the four functions that need to be implemented to solve a global optimization problem by simulated annealing. Note that these functions corresponds to the basic operations required by annealing (perturb and evaluate) plus two extra more: OnInitialize to initialize the solution, and the operator= that is useful whenever a solution needs to be copied from one variable to another one. It is important to mention that for some optimization problems, it may be inefficient to implement the operator= as this operator consumes a considerable amount of CPU time; for this cases other techniques to store and manipulate the solution may be used.

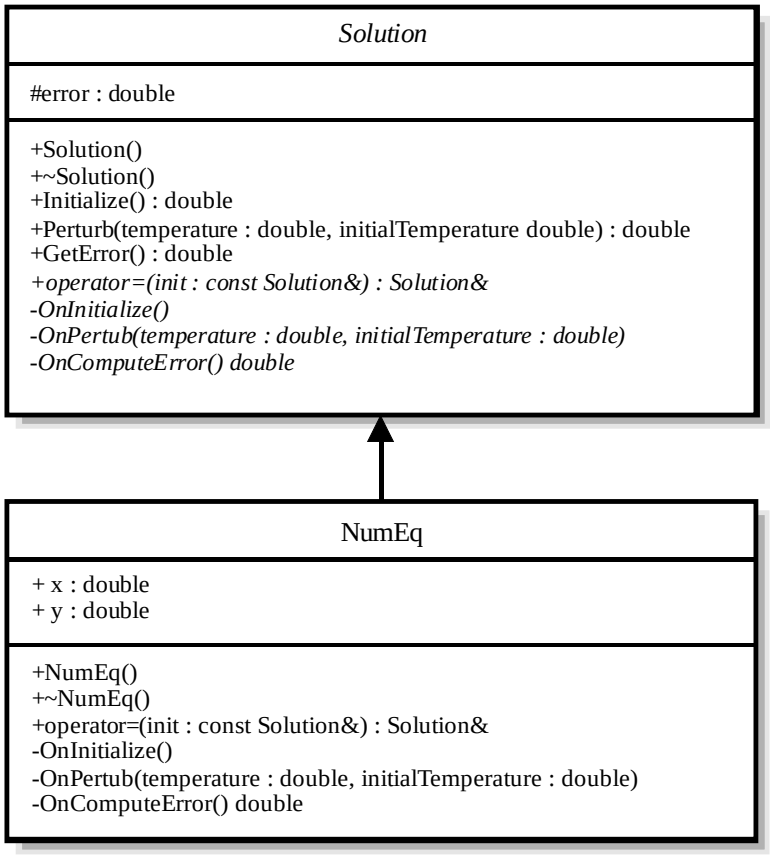


Fig. 5. UML diagram of a class to implement the solution of an optimization problem.

Figure 6 and 7 show a typical implementation using the C++ language for the Simulated Annealing class. The class may be implemented in others programming languages such as Java or C# with minor changes. Let now discuss briefly the implementation of this class.

There are several private functions on this class and are used only by the class itself. The function GetTemperature() is called every time the temperatures changes, its implementation is straightforward once the cooling scheduled has been defined; on the shown code there are two cooling schedules: exponential and linear. The function IsAcceptedByMetropolis() implements the metropolis algorithm of Equation 5, returns true when the perturbed solution must be accepted, and returns false otherwise. The function EstimateK() implements Equation 8. All the magic of the process is implemented in the function Anneal(), which is called several times if temperature cycling is used (i.e., the variable 'cycles' has a value bigger than one).
To use the SimulatedAnnealing class described, the function Start() must be called, this function requires three variables of the class Solution, namely 'solution', 'wk1' and 'wk2' ('solution' is the variable where the actual solution is stored; 'wk1' and 'wk2' are working solutions to perform the annealing process.) In the next section, it will be discussed how to use the SimulatedAnnealing class to solve a simple optimization problem.

SimulatedAnnealing.h

```
#pragma once
#include "Solution.h"
class SimulatedAnnealing
{
public:
	SimulatedAnnealing(void);
	~SimulatedAnnealing(void);
	int numTemps;
	int numIterations;
	double initialTemp;
	bool isCoolingScheduleLinear;
	int cycles;
	double Start(Solution& solution, Solution& wk1, Solution& wk2, double goal);
private:
	double GetTemperature(int index);
	bool IsAcceptedByMetropolis(double temperature, double deltaError);
	double Anneal(Solution& solution, Solution& wk1, Solution& wk2, double goal);
	double EstimateK(Solution& solution, int N);
	double finalTemp;
	double k;
};
```

Fig. 6. Header file using C++ to implement the SimulatedAnnealing class of Figure 4.

SimulatedAnnealing.cpp

```
#include "SimulatedAnnealing.h"
SimulatedAnnealing::SimulatedAnnealing(void)
{
	numTemps=100;
	numIterations=100;
	initialTemp=100.0;
	finalTemp=0.0001;
	isCoolingScheduleLinear=false;
```

```
        k = 10;
        cycles = 4;
}

SimulatedAnnealing::~SimulatedAnnealing(void)
{
}

double SimulatedAnnealing::Start(Solution& solution, Solution& wk1, Solution& wk2, double
goal)
{
        for(int i=0; i<cycles; i++)
        {
                if (Anneal(solution, wk1, wk2, goal)<=goal) break;
        }
        return solution.GetError();
}

double SimulatedAnnealing::EstimateK(Solution& solution, int N)
{
        double E = 1.0;
        double sum = 0.0;
        double sums = 0.0;

        for(int i = 0; i<N; i++)
        {
                E = solution.Perturb(initialTemp, initialTemp);
                sum+=E;
                sums+=(E*E);
        }
        double variance  = sums/(N-1) - (sum*sum)/(N*(N-1));
        return -log(0.8)*initialTemp/sqrt(variance);
}

double SimulatedAnnealing::Anneal(Solution& solution, Solution& wk1, Solution& wk2, double
goal)
{
        double error = solution.Initialize();
        if (error<=goal) return error; //We are alredy done. Unlikely!
        k = EstimateK(solution, 1000);
        wk1 = solution;
        wk2 = solution;

        finalTemp = goal;
        //
        bool hasImproved = false;
        double temperature, deltaError;
        int i;

        for (int n=0; n<numTemps; n++)
```

```
	{
		temperature = GetTemperature(n);
		hasImproved = false;
		//_____________________________________________ Iterate at this temperature
		for (i=0; i<numIterations; i++)
		{
			deltaError = wk1.Perturb(temperature, initialTemp) - error;
			if (IsAcceptedByMetropolis(temperature, deltaError))
			{
				wk2 = wk1;
				hasImproved = true;
				if (work1.GetError()<=goal) break;
			}
		}
		if (hasImproved==true) // If saw improvement at this temperature
		{
			wk1 = wk2;
			solution = wk2;
			error = solution.GetError();
			if (error<=goal) break;
		}
	}
	return solution.GetError();
}

bool SimulatedAnnealing::IsAcceptedByMetropolis(double temperature, double deltaError)
{
	if (deltaError=<0) return true;
	return Random(0.0, 1.0) < exp(-k*deltaError/temperature);
}

double SimulatedAnnealing::GetTemperature(int index)
{
	if (isCoolingScheduleLinear)
	{
		return initialTemp+index*(finalTemp-initialTemp) / (numTemps-1);
	}
	else
	{
		return initialTemp*exp(index * log(finalTemp/initialTemp) / (numTemps-1));
	}
}
```

Fig. 7. Source file using C++ to implement the SimulatedAnnealing class of Figure 4.

Figure 8 shows the header file for the Solution class using C++, Figure 9 shows the respective source file. As it can be seen from these figures, the Solution class is abstract as it has four abstract functions. Consequently, to create an object from the Solution class, a new derived class must be created and must implement: the operator=, OnInitialize(), OnPerturb() and OnComputeError(). Observe carefully the implementation of this class;

note that some functions are designed by pairs. For example, Perturb() calls internally the functions OnPeturb() and OnComputeError(). Similarly, Initialize() calls the functions OnInitialize() and OnComputeError(). As it will be seen in the next section using the Solution class to solve an optimization problem is pretty simple.

Solution.h

```
#pragma once

class Solution
{
public:
	Solution(void);
	~Solution(void);
	double Initialize(void);
	double Perturb(double temperature, double initialTemperature);
	double GetError(void);
	virtual Solution& operator =(const Solution& init) = 0;
protected:
	double error;
private:
	virtual void OnInitialize(void)=0;
	virtual void OnPerturb(double temperature, double initialTemperature)=0;
	virtual double OnComputeError(void)=0;
};
```

Fig. 8. UML diagram for a class to implement the solution.

Solution.cpp

```
#include "Solution.h"

Solution::Solution(void)
{
	error = 1.0;
}

Solution::~Solution(void)
{
}

double Solution::GetError(void)
{
	return error;
}

double Solution::Initialize(void)
{
	OnInitialize();
	error = fabs(OnComputeError());
	return error;
```

```
}

double Solution::Perturb(double temperature, double initialTemperature)
{
        OnPerturb(temperature, initialTemperature);
        error = fabs(OnComputeError());
        return error;
}
```

Fig. 9. UML diagram for a class to implement the solution.

## 8. Numerical example

Simulated annealing can be used to solve a broad range of optimization problems in artificial intelligence and other areas. However, it would be inappropriate to solve a complex problem to illustrate how to use simulated annealing. Thus, the two variable function of Equation 9 will be use for instructive purposes. Note that other optimization methods are more appropriate to solve this second order equation, and this section is only trying to set the basics for proper use of simulated annealing.

$$f(x, y) = x^2 + y^2 + 5xy - 4 \tag{9}$$

To get a better sense of the behavior of Equation 9, Figure 10 shows a plot of this equation. Let suppose that the goal is to find the values of x and y that minimize f(x, y). Clearly the solution is any point (x, y) that lies on the circle that intersects f(x, y) with the plane z = 0. Observe that simulated annealing is generally used when the solution has many variables, and finding or visualizing the solutions in these cases is much more difficult than interpreting the 3-D plot of Figure 10.

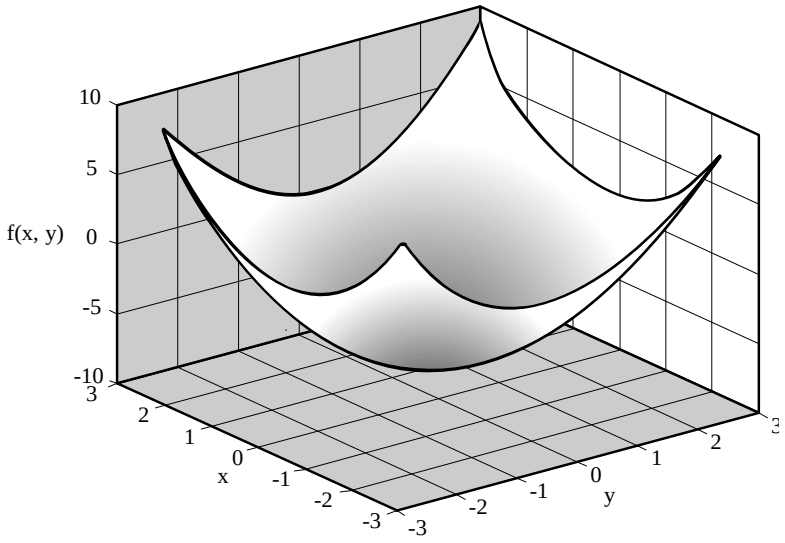


Fig. 10. A plot of Equation 9 as a function of x and y.

Consider Figure 11 that shows how simulated annealing works. At the beginning of the process the temperature is high (approximately 40 degrees in the figure) and the solution is perturbed so that it may lie on any of the points inside the dark circle. As the temperature decreases, the perturbation amount is reduced and the circle radius also decreases. At this temperature, the algorithm refines the quality of the solution previously found; note that solutions with high errors are not longer accepted (only solutions that reduce the error are accepted.) In other words, at low temperatures the algorithm moves the error down when the error is plot against x and y. Observe that the same number of iterations is used at each temperature while the exploring area is reduced; this increases the likelihood of finding the minimum.

Monitoring the progress of the process is important. For example, if the algorithm is not close to the minimum by the time the temperature has decreased considerably; simulated annealing will likely fail as the exploring area will be relatively too small. If this happens, it is better to restart the algorithm instead of performing useless iterations. One quick solution would be to increase the number of temperatures. Some will argue that the number of iterations should also be increased; however, it is important to note that it is not good idea to spend a lot of time at each temperature as the probability of acceptance will not change, if the temperature does not change either. Alternatively, if the temperature decreases slowly, the probability of the acceptance will gradually reduce, and the likelihood to accept a bad solution will be reduced as well.

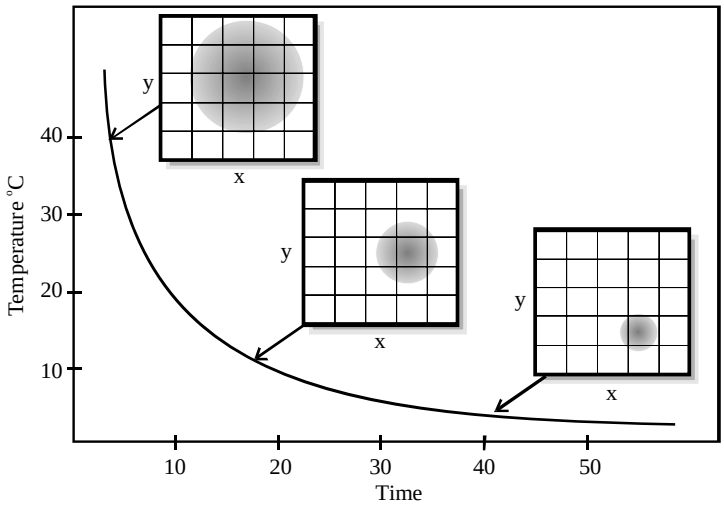


Fig. 11. Exploring area (represented as a gray circle) at each temperature.

Once a global view about how to minimize Equation 9, through the use of simulated annealing, has been presented, let actually show how to solve this particular problem. First, a new class derived from Solution must de created. Figure 12 and 13 show the header and source file for the class NumEq that is used to solve Equation 9; the respective UML diagram is shown at the bottom of Figure 5. Note that for this specific optimization problem, there are two variables of type double to store the solution 'NumEq.x' and 'NumEq.y'. The realization of the function OnComputeError() is straightforward as it directly implements Equation 9. The implementation of the function OnInitialize() is simple; it only assigns

random values between -10 and 10 to 'x' and 'y'. For other optimization problems, it is important to use common sense to implement the function OnInitialize(); if the problem does not provide enough information to do this, at least a valid initial value must be used.

Let know discuss the function Pertub() which is used to perturb the solution. Unfortunately, there are several ways to perturb a solution. It is recommended to try first the simplest way to perturb the solution; if this does not work, more sophisticate perturbation techniques may be used. Additionally, some practitioners prefer to perturb the solution a lot a high temperatures and reduce the degree the perturbation as the temperature decreases, this is what it is used on the example shown. However, it is important to mention that in some cases it is not possible to control the amount of perturbation, and for these cases the function Perturb() always applies the same amount of perturbation for each temperature.

By observing the function Perturb() in Figure 13, it can be observed that to perturb the solution, the value of 'x' is added to a random value which maximum amplitude is proportional to the current temperature. This method works really well, however, this approach may shift the solution too much, and 'x' may end in a region of invalid values. To alleviate this problem, an easy practice is to clip the solution values after perturbing. Alternatively, Figure 14 shows another way to perturb the solution; first a perturbation ratio is computed, then the new value of the variable is obtained by adding a proportional part of the old value plus a random variable; this method does not require clipping as the perturbation applied is blended naturally with the previous solution value. The authors have seen no evidence that one method is better than the other. However, it is important to note that when using the second method, the initial temperature is used only to compute the perturbation ratio and its value is not critical. Before leaving the discussion about how to implement the function Perturb(), please note that this function was specifically implemented using the knowledge that the values of 'x' and 'y' were in range from -10 to 10; other optimization problems may require a different implementation for this function.

The last function to discus is the operator=() which is used to copy a solution to another variable. This function must simply copy the solution variables from the source to the destination, specifically the variables: 'x', 'y' and 'error' in Figure 13.

NumEq.h

```
#pragma once
#include "Solution.h"

class NumEq : public Solution
{
public:
        NumEq(void);
        ~NumEq(void);
        double x, y;
        Solution& operator =(const Solution& init);
private:
        void OnInitialize(void);
        void OnPerturb(double temperature, double initialTemperature);
        double OnComputeError(void);
};
```

Fig. 12. Header file of the class NumEq to solve Equation 9.

NumEq.cpp

```
#include "NumEq.h"

NumEq::NumEq(void)
{
}

NumEq::~NumEq(void)
{
}

Solution& NumEq::operator =(const Solution& init)
{
	NumEq& eqInit = (NumEq&)init;
	x = eqInit.x;
	y = eqInit.y;
	error = eqInit.error;
	return *this;
}

void NumEq::OnInitialize(void)
{
	x = Random(-10.0, 10.0);
	y = Random(-10.0, 10.0);
}

void NumEq::OnPerturb(double temperature, double initialTemperature)
{
	x = x + Random(-temperature, temperature);
	y = y + Random(-temperature, temperature);
	//_____________ Clip values to avoid wandering too far
	if (x>10.0) x = 10.0;
	if (x<-10.0) x = -10.0;
	if (y>10.0) y = 10.0;
	if (y<-10.0) y = -10.0;
}

double NumEq::OnComputeError(void)
{
	return x*x+y*y+5.0*x*y-4.0;
}
```

Fig. 13. Source file of the class NumEq to solve Equation 9.

```
void NumEq::OnPerturb(double temperature, double initialTemperature)
{
	const double ratio = temperature/initialTemperature;
	x = (1.0 - ratio)*x + ratio*Random(-10.0, 10.0);
	y = (1.0 - ratio)*y + ratio*Random(-10.0, 10.0);
}
```

Fig. 14. Alternative method to perturb the solution.

Once a new class has been derived from Solution, it is possible to use the new class to solve the optimization problem of Equation 9. Figure 15 shows the actual code for the main function to do this. First, three variables of type NumEq are created; the variable 'solution' is where the final solution will be stored; 'wk1' and 'wk2' are working solutions. Next, a variable of type SimulatedAnnealing is created and configured, here, other configuration parameters may be set. The shown code sets only the initial temperature and the number of temperatures. Finally, the annealing process starts by calling the function Start(), and at the end, the values of 'x' and 'y' (which are the solution) are displayed.

Example.cpp

```
#include "SimulatedAnnealing.h"
#include "NumEq.h"

int _tmain(int argc, _TCHAR* argv[])
{
	NumEq solution, wk1, wk2;
	SimulatedAnnealing sa;
	sa.initialTemp = 10;
	sa.numTemps = 2500;
	cout<<"\r\nError = "<<sa.Start(solution, wk1, wk2, 0.00001);
	cout<<"\r\nx = "<<solution.x;
	cout<<"\r\ny = "<<solution.y;
	return 0;
}
```

Fig. 15. Main function to solve the problem of Equation 9.

Before moving into the next section, note that the classes SimulatedAnnealing and Solution are generics and can be used to solve any global optimization problem by simulated annealing.

## 9. Solving problems using simulated annealing

### 9.1 The traveling salesman problem

The traveling salesman problem is a classical problem in artificial intelligence, where a seller has to visit N cities that are located at given positions, and finally he has to return to his city of origin (Press et al., 2002). For this problem, each city has to be visited only once and the resulting path should be as short as possible. Press et al. shows actual code using the C++ language to solve this problem and provides several tips worth trying to set the parameters of the algorithm. There, the problem is solved using the two basic operations described in this chapter: perturb and evaluate. The operation of perturb is performed by using two different type of perturbations. To evaluate the quality of the solution the path length is used. Press et al. ends the subject of simulated annealing by introducing a hybrid algorithm using the downhill simplex method.

### 9.2 The N-Queens problem

The N-Queens problem is a famous problem that has been attacked by a wide variety of search algorithms (Jones, 2005). It is defined as the placement of N queens on an N-by-N

board such that no queen threatens any other queen using the standard rules of chess. This problem may be planned and solved by simulated annealing (Jones, 2005). The method used by Jones is similar to the method proposed here; however, our method is object-oriented and promotes code reuse.

### 9.3 Artificial neural network training

In (Masters, 1993), it is suggested to use simulated annealing for neural network training. Masters suggest a hybrid algorithm that combines simulated annealing with typical gradient based algorithms. Simulated is used only for initialization, and gradient based algorithms are used to refine the quality of the solution. Additionally, Masters suggest using simulated annealing in combination with other deterministic methods, for example regression to estimate the output weights of a neural network and perturb only the hidden weights. For artificial neural network training the implementation of simulated annealing requires a good random number generator, see (Press et al., 2002) to see code to implement such type of generators. The authors have suggested simulated annealing using temperature cycling for neural network training (Ledesma et al., 2007).

The free software Neural Lab is a powerful tool to simulate artificial neural networks, and it can be downloaded from http://www.fimee.ugto.mx/profesores/sledesma/. Neural Lab implements simulated annealing for neural network training using temperature cycling and several hybrid algorithms.

## 10. Conclusions

Simulated annealing is a powerful algorithm to solve global optimization problems. It has been successfully used in artificial intelligence (Russel & Norvig, 2002), and real life problems that do no have an appropriate model. There are still many aspects of simulated annealing open for research, including how to reduce the running time of the algorithm, how to optimize the cooling schedule and how to adapt the algorithm as the temperature and error change. The authors have presented several practical considerations that will help the reader to use simulated annealing to solve real life problems.

## 11. References

Jones, M. T. (2005). *AI Application Programming* (2nd edition), Charles River Media, ISBN 1-58450-421-8, Massachusetts, U.S.A.

Luke, B. T. (2007). Simulated Annealing Cooling Schedules, available online at http://members.aol.com/btluke/simanf1.htm, accessed June 1, 2007.

Ledesma, S., Torres, M., Hernandez, D., Avina, G. & Garcia, G. (2007). Temperature Cycling on Simulated Annealing for Neural Network Learning, Proceedings of MICAI, pp. 161-171, ISBN 978-3-540-76630-8, Mexico, November 2007, Springer-Verlag Berlin Heidelberg, Aguscalientes.

Masters, T. (1993). *Practical Neural Network Recipes in C++*. Academic Press, Inc., ISBN 0-12-479040-2, California, USA – London, UK.

Masters, T. (1995). *Advanced Algorithms for Neural Networks.* John Wiley & Sons Inc., ISBN 0-471-10588-0, New York, USA.

Press, W. H., Teukolsky, S. A., Vetterling, W. T., & Flannery, B. P. (2002). *Numerical Recipes in C++: The Art of Scientific Computing* (Second Edition), Cambridge University Press,

ISBN 0-521-75033-4, Cambridge, New York, Melbourne, Madrid, Cape Town, Singapore and Sao Paulo.

Reed, R. D. & Marks II, R. J. (1999). *Neural Smithing: Supervised Learning in Feedforward Artificial Neural Networks*, the MIT Press, ISBN 0-262-18190-8, Massachusetts, USA.

Russel, S. J. and Norvig, P. (2002). *Artificial Intelligence: A Modern Approach (2nd edition)*, Prentice Hall, ISBN 81-203-2382-3, New Jersey, U.S.A.