Spring 2013

Excitation Optimization of a Standard LRC Circuit with Impulsive Forces Selected via Simulated Annealing

Ryan M. Spies
Hamline University

Excitation Optimization of a Standard LRC Circuit with Impulsive Forces Selected via Simulated Annealing

Ryan M. Spies

An Honors Thesis

Submitted for partial fulfillment of the requirements for graduation with honors in Physics from Hamline University

April 26, 2013

Abstract

For an unknown oscillator, it is sometimes useful to know what the potential energy function associated with it is. An argument for using a method of determining the optimal sequence of impulsive forces in order to find the potential energy function is made using principles of energy. Global optimization via simulated annealing is discussed, and various parameters that can be adjusted across experiments are established. A method for determining the optimal sequence of impulsive forces for the excitation of a standard LRC circuit is established using the methodology of simulated annealing.

Contents

List of Figures

Chapter 1

Motivation

We want to develop a methodology that will allow us to determine the optimal sequence of pulse timings for an oscillator, which will in turn determine the optimal motion for the oscillator as well. This methodology must also allow us to find the potential energy function of a given oscillator. In order to understand what it means to determine the optimal sequence of pulse timings, we must first understand the dynamics of oscillators. In particular, we want to understand the dynamics of oscillators after they are hit by an impulsive force. Also, we want to know what it means to find the potential energy function for a given oscillator, and to figure out how that is related to the problem of finding the optimal sequence of pulse timings. Finally, we want to understand what it means to take the principles of these classical oscillator problems and apply them to electronic oscillators.

For a thought experiment, let us consider a pendulum. If we gave this pendulum a slight push, and if there is no friction present, then the pendulum will move and the motion will not decrease. If we want to get this pendulum to swing out further we want to push it more, but then comes the matter of determining when it is best to push the pendulum again. Assuming that the pendulum can only be pushed from one direction, the best moment for pushing it again would be when it is back at its equilibrium position and when it is going in the same direction as the push. However, since the pendulum has a periodic behavior that is nonlinear for sufficiently large angles (so that the approximation for the angular position θ of the pendulum, $\theta \approx \sin(\theta)$, or the angles for which simple harmonic motion occurs, no longer holds) the timing of these pushes becomes a problem. The amount of time that one would have to wait before pushing the pendulum again at the best time changes, so this motivates a method that can find when to push the pendulum without prior knowledge of the pendulum's behavior.

In order to understand how our thought experiment relates to the larger problem let us consider the dynamics of oscillators in general, and start with the example of the simple harmonic oscillator.

1.1 Simple Harmonic Oscillator

The motion of the simple harmonic oscillator can be determined by first finding the sum of forces that are acting on the oscillator. For the simple harmonic oscillator there is only one force acting on it. This is the force due to the spring attached to the mass. Our sum of forces equation can now be written as:

$$\Sigma F = -kx = m\ddot{x}, \tag{1.1}$$

In equation 1.1 F is the force, k is Hooke's constant for that spring, m is the mass of the oscillator, and x is the one-dimensional position of the oscillator. The dots represent the second derivative with respect to time t. We can use equation 1.1 to find the equation of motion:

$$m\ddot{x} = -kx. \tag{1.2}$$

This is then rewritten to obtain:

$$m\ddot{x} + kx = 0. \tag{1.3}$$

Equation 1.3 is a homogenous ordinary second-order differential equation that can be solved to find x. In classical mechanics we can also refer to this equation as the unforced case for the simple harmonic oscillator. Using methods for solving ordinary differential equations the position x as a function of time can be determined in terms of the various constants that govern the differential behavior of the oscillator.

However, this is not the case of the oscillator that we want to investigate in this experiment. Instead, we want to investigate a particular instance of the forced oscillator where the oscillator is driven by a sequence of impulsive forces. The differential equation for this case is:

$$m\ddot{x} + kx = F\tau(t). \tag{1.4}$$

Here F is a constant in units of force that represents the magnitude of the impulsive force, the timing function, $\tau(t)$, is a unitless function that is written as a series of Dirac delta functions:

$$\tau(t) = \delta(t_0) + \delta(t_1) + \delta(t_2) + ... \tag{1.5}$$

In equation 1.5 the moments in time t_i (where $i = 0, 1, 2, ...$) are when an impulsive force occurs. It is important to note that this particular function is a model for showing when impulsive forces happen. For any physical examples an impulsive force is simply a force that delivers a lot of momentum over a short length of time in comparison to the overall duration of the motion that is considered.

The ultimate goal of the experiment is to find the series of optimal t_i for exciting an oscillator. In order to determine at which moments an impulsive force needs to act on an object for obtaining the optimal response, we must first consider the effect that an impulsive force has on the overall momentum of an object. If we integrate the impulsive force over time we get that the impulse that is added to the oscillator is:

$$\int F\, dt = F\Delta t.$$

Here Δt is the length of a brief interval in time during which the impulsive force acts.

When the impulsive force F is delivered the final momentum p_f of the oscillator will then be:

$$p_f = p_i + F\Delta t, \tag{1.6}$$

where p_i is the initial momentum at the moment of the impulse. Both sides can be rewritten in terms of energy. We can then consider the final energy E_f of the oscillator after the impulsive force gives the system an amount of energy u_i to be:

$$E_f = \frac{p_i^2}{2m} + u_i. \tag{1.7}$$

In order to optimize the motion of the oscillator, the energy u_i that is delivered to the oscillator must be maximized. In order to determine how to maximize this energy we first take a look at the change in energy that occurs due to the impulsive force:

$$\Delta E = \frac{p_f^2 - p_i^2}{2m} = \frac{p_i^2 + 2F\Delta t p_i + F^2\Delta t^2 - p_i^2}{2m},$$

which we rewrite as:

$$\Delta E = \frac{2F\Delta t p_i + F^2\Delta t^2}{2m}. \tag{1.8}$$

For our one-dimensional case we can leave equation 1.8 as is and recognize that when the initial momentum p_i is at its maximum then the change in energy is maximized as well. For two or three dimensional problems we can use vector algebra to rewrite 1.8 as:

$$\Delta E = \frac{2\Delta t\vec{F}\cdot\vec{p_i} + F^2\Delta t^2}{2m}. \tag{1.9}$$

In equation 1.9 it is apparent that in order to maximize the change in energy, the dot product of the impulsive force and the initial momentum at the moment of the impulsive force must also be maximized. This means that in order to excite an oscillator with impulsive forces, those forces must also be applied in such a way that they will be in the same direction as the initial momentum at the time of the impulsive force. This generalization to two or three dimensions applies to our one dimensional problem as well.

Given the conditions that must be met in order to maximize the momentum after the delivery of an impulsive force, we can determine the sequence of impulsive forces that will optimize the motion of our oscillator. We first solve equation 1.3 for x. In order to do this we will first define a new quantity ω_0, or the resonant frequency of the oscillator:

$$\omega_0 = \sqrt{\frac{k}{m}}.$$

We can use this substitution to rewrite equation 1.3 as:

$$\ddot{x} + \omega_0^2 x = 0. \tag{1.10}$$

The general solution to this is:

$$x(t) = A\sin(\omega_0 t + \phi). \tag{1.11}$$

In equation 1.11 A is some maximum amplitude, and ϕ is some change in phase. These constants can be determined using initial conditions. From this point on we will consider an amplitude of one and a phase change of zero. We must now consider finding when the velocity is maximized, which can be determined by finding the acceleration. We take the first and second derivatives to find the velocity and acceleration of the simple harmonic oscillator respectively.

$$\dot{x} = \omega_0 \cos(\omega_0 t) \tag{1.12}$$

$$\ddot{x} = -\omega_0^2 \sin(\omega_0 t) \tag{1.13}$$

We find the first point at which equation 1.13 is zero and equation 1.12 is positive in order to find the point in time at which the velocity, and therefore the momentum is maximized. Our system will look like the following:

$$0 < \omega_0 \cos(\omega_0 t) \tag{1.14}$$

$$0 = -\omega_0^2 \sin(\omega_0 t) \tag{1.15}$$

The system of equations is solved to obtain that the first moment in time where both of these conditions are met is the time when the oscillator completes the first full period. We refer to this time as t_p, which we can write out as:

$$t_p = \frac{2\pi}{\omega_0}.$$

By the conclusions that we arrived at earlier with equations 1.8 and 1.9, the optimal timing function is:

$$\tau(t) = \delta(t_p) + \delta(2t_p) + \delta(3t_p) + ... \tag{1.16}$$

1.2 General Oscillator

Now that we have established the basic premise for maximizing the response of the linear oscillator, let us consider a more general case where instead of behaving like a linear spring there is a non-linear response. Before we can do this, we must first consider the simple harmonic oscillator in terms of a potential energy function so we can understand the general oscillator problem. In general we can write the potential energy function U of a conservative force as:

$$U(x) = -\int \vec{F}(x) \cdot d\vec{x}. \tag{1.17}$$

For our simple harmonic oscillator we can find the potential energy function of the spring using equation 1.17.

$$U(x) = -\int -kx \; dx$$

$$U(x) = \frac{1}{2}kx^2. \tag{1.18}$$

We also get from equation 1.17 that the relationship between a conservative force and a potential energy can be written as:

$$\vec{F} = -\vec{\nabla}U, \tag{1.19}$$

or in one dimension:

$$F = -\frac{dU}{dx}. \tag{1.20}$$

For an oscillator whose motion is determined by a conservative force, assuming no friction, we get that the equation of motion is:

$$m\ddot{x} + \frac{dU}{dx} = 0. \tag{1.21}$$

Sometimes for the case of the general oscillator, and in particular when nonlinear behavior is present in the motion of the oscillator, we do not always know the potential energy function. However, there is a relationship between the amplitude of an oscillator and the kinetic energy that is given to an oscillator that can be exploited in order to determine the potential energy function of the oscillator. If we give an oscillator a discrete amount of energy each time and then we record the maximum amplitude attained by the oscillator then we can use these data points to fit a curve for the potential energy.

In order to show that we can do this, we first look at the general equation for the total energy in a system. We define the kinetic energy K as:

$$K = \frac{p^2}{2m}$$

Our total energy is then:

$$E = \frac{p^2}{2m} + U. \tag{1.22}$$

The total energy in any system can be expressed as a sum of the kinetic and potential energy at any moment in time. Now we examine the energy at a particular point in time. If there is a discrete amount of energy that is present in a system, and if we can assume that none of that energy is lost to friction, then we can conclude that there are moments in time where all the energy is kinetic energy and where all the energy is potential energy. For the moments where all the energy is kinetic we know that:

$$E = \frac{p_{max}^2}{2m}. \tag{1.23}$$

We can conclude this because at this moment the momentum of the system is maximized since all of the energy present in the system is kinetic. We also know from the work in the previous section that at the moment the momentum is maximized, that it would be the best time to deliver an impulsive force to further increase the motion of the oscillator. This establishes that we require a method that determines the optimal timing in order to maximize the change in energy, and to make sure that the change in energy with each impulsive force is the same.

In order to determine the potential energy function from an experiment where the optimal sequence of impulsive forces is determined, we must also examine what happens at moments when the total energy is all potential energy. In an oscillator that assumes that the effects of friction on the total energy is negligible, when we have established that after an impulsive force the kinetic energy that can be present after that force is at a maximum, we know then

that the potential energy at a later point will also be at a maximum. Stated mathematically this is:

$$E = U. \tag{1.24}$$

At these moments the oscillator is at it's maximum position, which allows us to rewrite equation 1.24 in terms of position as:

$$E = U(x_{max}). \tag{1.25}$$

This means that in order to find the potential energy as a function of position, an experiment must also include a way to measure the position after an impulsive force is delivered. After gathering the maximum positions after each impulsive force the total energy as a function of position can be determined via numerical methods such as interpolation, and this function is equivalent to the potential energy function as a function of position.

1.3 LRC Extensions

This experiment will utilize a LRC circuit as the oscillator. In order to understand the behavior of this circuit, it is first important to note that there is a frictional element present in the system that removes energy. In order to understand the impact that a frictional element will have on the overall motion of the system, it is important to consider the full equation for the damped harmonic oscillator:

$$m\ddot{x} + b\dot{x} + kx = 0. \tag{1.26}$$

In this equation the value b corresponds to a drag constant, which determines by how much the corresponding drag force removes energy from the system. The drag force for this classical case is the friction due to the surrounding medium.

We now consider the equation for the voltage behavior of a charged LRC circuit:

$$L\ddot{q} + R\dot{q} + \frac{1}{C}q = 0. \tag{1.27}$$

Here L is the inductance, R is the resistance, C is the capacitance, and q is the charge. From here we can begin to make a few analogies that tie the LRC to the classical damped harmonic oscillator problem, and how to determine energies and potential energy functions.

To begin with L corresponds to m, which also gives us the following analogy for kinetic energy K in an electronic system;

$$K = \frac{1}{2}LI^2.$$

Here I is the current, or the first time derivative of charge. The charge q, and its derivatives take on the role of the position x in the electronic system.

In the classical damped oscillator problem, we can determine the work W that is removed from the system by evaluating the following integral:

$$W = \int b\dot{x}\, dx.$$

It is important to note that we can rewrite the differential element dx in terms of time. This can simply be achieved by taking the first derivative of x and then multiplying it by dt. This gives us the new integral:

$$W = \int b\dot{x}^2 \, dt. \tag{1.28}$$

Therefore, if we can obtain x as a function of time, a derivative of that can be taken and this integral can be solved to keep track of how much work was eliminated due to friction as time goes on. By our analogy with the LRC, R takes on the role of b. This means that for our electronic analog we get for the work removed from the system over time:

$$W = \int RI^2 \, dt. \tag{1.29}$$

Finally, we have the matter of determining the potential energy function. We continue our analogy to note that k is related to the inverse of C. The potential energy function for our system is then:

$$U = \frac{q^2}{2C}. \tag{1.30}$$

This is related to the electrical potential V by dividing by q.

Now we must determine how to find the electrical potential from our proposed experiment as a function of the charge q. We must first be able to determine the charge at any point during the experiment. For this, we consider the charging behavior of the LRC circuit, and from there we can determine what the charge present is at any point. We now consider a particular case of the LRC where:

$$L\ddot{q} + R\dot{q} + \frac{1}{C}q = V_0\tau(t). \tag{1.31}$$

Here V_0 is a constant value for the potential. This driven case of the LRC circuit is the electronic form of our driven harmonic oscillator problem, only with some frictional element included. When the circuit is driven by a voltage impulse, it is considered to be a pure DC circuit. This means that the effects of the inductor can be ignored while it is in this state. We obtain that the voltage as a function of time for while the voltage impulse is present is then:

$$V(t) = V_0(1 - e^{-t/RC}). \tag{1.32}$$

Since the charge present in a capacitor is found by:

$$q = VC,$$

It follows that:

$$q(t) = CV_0(1 - e^{-t/RC}), \tag{1.33}$$

when the capacitor is charging. As the capacitor discharges over time from the total amount of charge q_0 we can state that the total charge present is then:

$$q(t) = q_0 e^{-tR/2L} \cos(\frac{t}{\sqrt{LC}}), \tag{1.34}$$

which is the to equation 1.31 for the appropriate boundary conditions.

When we have an impulsive voltage, we can determine what the charge will be after a given point in time. Let us suppose that before an impulsive charge is given, that the total charge present is some discrete amount q_i. We then get that for after an impulsive charge is delivered:

$$q_f = q_i + CV_0(1 - e^{-t_i/RC}), \tag{1.35}$$

where q_f is the final amount of charge present and t_i is the duration of the impulse. If measurements are taken after that, then the charge present at the time of a particular measurement will be:

$$q(t) = q_f e^{-tR/2L} \cos(\frac{t}{\sqrt{LC}}). \tag{1.36}$$

For an experiment where the voltage is measured, and the time at which the measurement was taken can be obtained, then the charge at this point can also be found. This allows to find the electric potential as a function of charge, and by extension will allow us to find the potential energy function as a function of charge as well.

Chapter 2

Methodology

For our experiment, the oscillator of choice is an RLC circuit driven by a sequence of impulsive charges that are controlled by the Arduino microcontroller. Our optimization algorithm, implemented in Python version 2.6, is a simulated annealing algorithm that allows the user to control various stages of the algorithm across experiments. Both versions for simulations and for the actual experiment are developed, with the code of each version located in the appendices.

2.1 Simulated Annealing

The method used for selecting the sequence of pulses used to excite the oscillator is simulated annealing. Simulated annealing is an optimization method that mimics the thermodynamic process of annealing with the goal of finding a particular global optimum within a solution space. This solution space varies from implementation to implementation. A general process for simulated annealing for finding a global minimum is outlined in the pseudocode example in Figure 2.1. Upon initialization, the algorithm makes a guess x from the problem's solution space and computes the value of the fitness function $f(x)$ (also referred to as the cost function) for that particular guess. In addition to that a temperature T that represents how much energy is available at the start of the algorithm is determined.

After the initial guess a loop is begun that starts by generating a neighborhood, or a subset of the solution space that is centered about the current accepted guess. A new guess is selected from this neighborhood, and subsequently the fitness of this new guess is computed as well. The difference between the two fitnesses Δf is then calculated as per the calculation of the quantity change in figure 2.1. For the algorithm in the pseudocode example, if this change is negative the new guess is always accepted. If the change is non-negative, in order to encourage exploration of the solution space within the algorithm the chance of accepting this "worse" guess is found using a Boltzmann-like probability distribution as described in the following equation:

$$P(\Delta f) = \exp\left(\frac{-\Delta f}{kT}\right) \tag{2.1}$$

```
procedure simulated annealing (T, solution space)
select guess from solution space
fitness = f(guess)
loop until T < termination condition:
    generate neighborhood about guess
    select newGuess from neighborhood
    newFitness = f(newGuess)
    change = newFitness-fitness
    if change < 0:
        accept newGuess
    elif exp(-change/kT) > random(0,1):
        accept newGuess
    decrease T
return best guess and best fitness
```

Figure 2.1: Pseudo-Python implementation of the standard simulated annealing algorithm for finding a global minimum.

In equation 2.1 k is the Boltzmann constant. In general, the algorithm can have other solution selection methods other than the one defined in our example. After determining whether or not the guess is accepted, the temperature is decreased and the loop repeats until the temperature is cooler than the termination condition. By the end of the algorithm the best guess that it made is returned along with the corresponding fitness. This algorithm can also be adjusted for finding the global maximum of a particular solution space by always accepting the new guess if the change is positive, and by changing the negative sign in equation 2.1 to a positive sign.

There are a wide variety of things that can be altered about this simple algorithm. These include determining the chance of keeping a guess, how to decrease the temperature, how to select the neighbourhood for a particular guess, and initializing the relevant constants for these. An example of a flexible algorithm in Python that allows a user to make the changes mentioned in the following sections can be seen in Appendix A.

2.1.1 Acceptance Probability

For the typical simulated annealing algorithm, a probability selection method that comes from another algorithm called the Metropolis algorithm is used. For the problem of finding the global maximum of a particular solution space the probability P of selection is defined as:

$$P(x, y, T) = \begin{cases} 1, & y > x \\ \exp\left(\frac{y-x}{kT}\right), & y < x. \end{cases} \tag{2.2}$$

In equation 2.2, x is the previous guess and y is the most recent guess. This is a mathematical representation of the guess selection process outlined in figure 2.1, and this the most frequently used guess selection probability method in simulated annealing. In addition to that method, a slightly modified probability distribution method was used whose Python implementation can be found on line 47 of Appendix A. Mathematically, this method can be written as:

$$P(x, y, T) = \begin{cases} 1 - \exp\left(\frac{x-y}{kT}\right), & y > x \\ \exp\left(\frac{y-x}{kT}\right), & y < x. \end{cases} \tag{2.3}$$

For the method defined by equation 2.3 the probability of keeping a guess better than the previous one is not one, but instead one minus the probability of choosing a worse guess that has the same absolute value of the difference in fitness from the previous guess. This allows for some variety in the experiments that can be performed.

2.1.2 Annealing Schedule

The annealing schedule (also known as the cooling schedule) is the way by which the algorithm decreases the temperature for each run of the loop. The two methods most commonly used are the exponential annealing schedule and the linear annealing schedule.

The exponential annealing schedule causes the temperature to decay exponentially over the course of the algorithm. A constant c between 0 and 1 (typically 0.95 for most implementations) is chosen, and at the end of each run of the loop the current temperature is multiplied by this constant. This method of decreasing can be written as:

$$T_{new} = cT_{previous}. \tag{2.4}$$

The linear annealing schedule causes the temperature to decrease linearly over the course of the algorithm. An amount ΔT is subtracted from the temperature as the algorithm progresses, and this rule for determining the temperature can be mathematically written as:

$$T_{new} = T_{previous} - \Delta T. \tag{2.5}$$

The algorithm in Appendix A allows for a choice between the exponential and linear schedules as defined by the function on line 160, but the main experiment utilizes an exponential schedule with a constant c of 0.95.

2.1.3 Neighborhood Selection

The last thing that is varied in the simulated annealing algorithm is the neighborhood selection method. This is the method by which the algorithm finds new solutions to test. In a typical simulated annealing algorithm the neighborhood is always a constant interval centered about the current accepted guess. However, if the neighborhood is adjusted over the course of the algorithm to decrease in size as the temperature decreases, then the algorithm performs better than if the algorithm has a fixed neighborhood size. This was established in the paper "Dynamic Neighbourhood Size in Simulated Annealing" by Xin Yao.

There are several different methods for neighborhood selection that are utilized in the algorithm that is written up in Appendix A. The first of these is simply using a constant interval that is recreated every time that the algorithm restarts. The programmer selects some number that represents the distance away from the current guess in the solution space in both directions (for a one

dimensional space), and then the algorithm selects a different solution to test from this neighborhood.

The other approaches all rely on functions that decrease in size as T decreases. Each of these approaches uses another user-defined constant, m, which for this set of experiments has the experiment start by exploring a neighborhood that is one quarter of the size of the solution space. One of these options is recreating the interval bounds at each time as a function of T proportional to the square root of T as follows:

$$N = \sqrt{\frac{T}{m}}, \tag{2.6}$$

where N is the size of the neighborhood.

There are a few other functions that have similar limit behavior to the square root function as T decreases. The reason for these functions is to test an algorithm where the neighborhood decreases as T decreases, or in terms of limits:

$$\lim_{T \to 0} N(T) = 0. \tag{2.7}$$

This motivates us to explore a few other functions that have this behavior. In the algorithm in Appendix A three additional functions for determining the limits of the neighborhood that have this limit behavior as defined in equation 2.7. First, one of these has a linear proportionality with respect to T, or simply that:

$$N = mT. \tag{2.8}$$

Secondly, another method is proportional to T^2, or as defined within the algorithm:

$$N = \frac{T^2}{m}. \tag{2.9}$$

The last of these methods uses a decreasing exponential function. This function as defined within the algorithm is:

$$N = me^{-\frac{1}{T}} \tag{2.10}$$

The last method for neighborhood selection was one where the neighborhood was selected so that it was increasing each time. This was meant both to see what the algorithm behavior would be if the neighborhood increased as a function of T and to test what would happen if at the end the entire solution space was available for the algorithm to choose from. In the physical experiment, this method was not selected. The function that is defined for this is:

$$N = \frac{m}{T\sqrt{2\pi}} e^{-\frac{1}{2T^2}}. \tag{2.11}$$

The user is capable of selecting any of these neighborhood selection methods at the beginning of the algorithm.

2.2 Electronics

2.2.1 Circuitry

The circuit tested was a simple LRC circuit, but in order to be able to manipulate and interact with this circuit via computer control the Arduino Uno

microcontroller had to be used. This placed some limitations on the sort of circuit that can be constructed.

First, the frequency of the LRC was limited because the Arduino can only sample once every 100μs. In order to gain a sufficient range of data from which an optimal value can be obtained, the parts were selected as follows:

$$R = 10\Omega, L = 3.9\text{mH}, C = 6.5\mu\text{F}.$$

In addition, there were limits to the sorts of voltages that could be obtained by the Arduino. The Arduino uses an on board analog to digital converter that can only sample from 0V up to 5V. Therefore, negative voltages make it so that we must design a circuit that will always give a positive voltage. In order to take advantage of the full capability of the Arduino, this meant that the LRC had to be set at a reference of 2.5V. This means that instead of the LRC going to ground, it instead goes to a voltage supply that will remain at a constant 2.5V. In order to drive the circuit with pulses, another voltage supply was needed that could switch between providing 5V and 2.5V, and that this switching behavior could also be controlled via the Arduino.

In order to provide voltage supplies that could provide the necessary voltage behavior and the currents that were needed for this particular circuit, a couple of operational amplifiers (LF741) were used to make two follower circuits. The resulting circuit using these follower circuits is diagrammed in Figure 2.2.

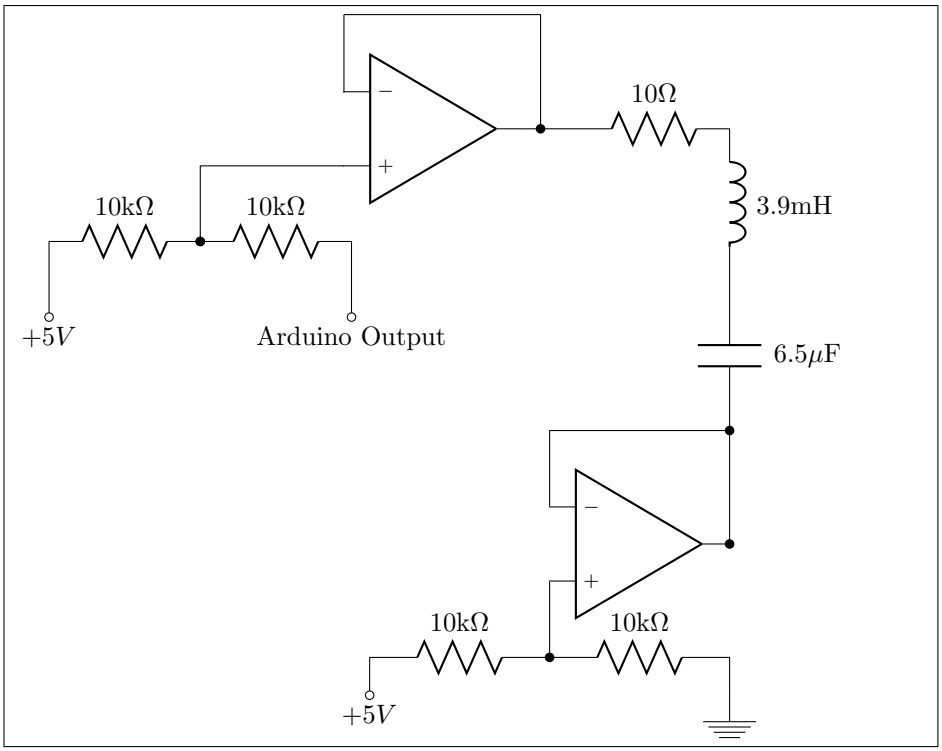

Figure 2.2: Circuit diagram of the follower circuit attached to the RLC circuit.

A follower circuit is one that keeps track of the voltage on one terminal, and then outputs that exact same voltage. On the positive input is a voltage divider

that is selected so that the voltage at that terminal is 2.5V. For the follower circuit that was used for providing a constant reference voltage this consisted of two 10kΩ resistors, one coming from a constant supply of 5V and another attached to ground. The positive input is connected to the place on the voltage divider where the voltage is 2.5V. For the follower circuit that provided the pulses to drive the circuit, the other end of the voltage divider was attached to a digital output from the Arduino. This output would be at logic 0, or at ground, for most of the time, and would be at logic 1, which is 5V for this Arduino, when a pulse is given. Once the positive inputs are established, a connection between the negative input and the output of the op-amp is used. The results are two circuits that behave like the voltage sources that are needed for this particular problem.

The two ends of the LRC were then attached to the outputs of the follower circuits. The Arduino could then be used to monitor the voltage along any point of the circuit.

2.2.2 Arduino Program

The final part of the circuitry was to set up the Arduino properly with the correct program. As a microcontroller, the digital electronics within are controlled by programs (called sketches) that are written by the user using C++ and some special libraries therein that are specific to Arduino programming. The full Arduino sketch can be seen in Appendix D.

The program begins with the setup section, which determines some behaviors of the microcontroller that will only be executed at the beginning of the sketch. In this instance, the serial port is set to as high a speed as a computer can handle, and a pin is selected to be an output for driving the LRC. After that a function called pulse() is established, which takes an output pin and turns it on for a length of time before shutting off again.

The second part of the program is one that will repeat in a loop, meaning that this can be used to run multiple experiments. First, the microcontroller waits for something to be available on the serial port. From the python program this string will be something that looks as so:

$$3[100, 200, 300]20.$$

The first number in the incoming string is interpreted to be the number of timings between pulses, and in this case there are three timings. Then the numbers contained within the brackets are the timings between particular pulses in microseconds, which in this case is 100, 200, and 300 respectively. The last number is interpreted as how many points of data should be obtained for this particular run.

After the numbers are interpreted the program then initiates a loop structure in which a pulse of five microseconds occurs, and then it cycles through an array of timings to figure out how long to wait before the next iteration of the loop. Once this loop is complete one final pulse is given. Then the program immediately gathers data that is sent back to the computer for interpretation.

2.3 Main Python Program

In appendices B and C are two parts of a program that is used to interact with the Arduino on the computer side. Appendix B is the main simulated annealing program, and Appendix C is a module that defines a function that allows the program to interact with the Arduino.

The program starts off by defining a lot of the same functions that were defined in the algorithm in Appendix A. After that is where the programs start to differ. Instead of allowing the user to reset the temperature at will, the temperature for each run of the simulated annealing algorithm will always start at 10. The user is also unable to adjust the constant k for selection probability and the constant m for each neighborhood selection method. The termination condition is also fixed for this particular program. The user is allowed to vary among various neighborhood selection methods, acceptance probability calculation methods, file names, and data points gathered for the fitness function.

In this program, simulated annealing is used in nested loop structures to build a pulse train, or a sequence of pulses with particular timings associated with each of them. The user specifies under the "how many pulses are in the pulse train" portion of the program how many timings in between pulses will be specified.

The first loop structure will define how many times the experiment is repeated. If the user only wants to run the experiment once, they will only run the experiment once. Otherwise, they can specify a nonzero integer amount of times to repeat the experiment for.

The second loop structure within the first loop is the one that finds the solution for a particular number of pulse timings j. It begins by making an initial guess for the jth element as defined by the loop iterator, and then determining the fitness. The graph and spreadsheet for this particular element of the solution is also created within this loop structure.

The final loop structure that is within the second loop is the actual simulated annealing algorithm itself. The simulated annealing process is repeated for each element of the pulse train. This allows for the solution to be built up element by element instead of all at once.

Chapter 3

Results

3.1 How to Interpret Simulation Results

These results of this simulation are from 32 different runs of the algorithm presented in Appendix A. Figure 3.1 is the graph of the fitness function that this algorithm explored. The temperature is the given value of T at the start of the run, and the Boltzmann Constant was the given value of k for that experiment. The interval weight was a value of m that was selected for the various methods of neighborhood selection that were discussed in the methodology section. This algorithm allowed for the selection of a neighborhood selection method, an annealing schedule, and the method by which the selection probability was determined. Relevant annealing schedule constants were also chosen for each run, and the termination temperature was also set. The algorithm also allows for a simple gaussian test to be chosen to determine if a certain set of options will select the optimal input that is expected for that function, and to select a different interpolated function generated from a random set of points for testing to see if certain options will converge on the global maximum or simply on a local maximum. Upon completion, the algorithm returns a graph, a spreadsheet of the data plotted on the graphs, and a logfile that talks about the choices made for that particular run. This set of simulations did not explore the linear method of selection, and no other annealing constant other than 0.95 was chosen for any given run. The Boltzmann Constant was also unchanged across runs, and the termination temperature condition did not vary from run to run. These choices were made so that a greater emphasis could be placed on exploring neighborhood selection and selection probability, and how varying these two conditions affects how a particular run will turn out. At the end of each run, it is then determined whether or not the particular run converged on the global optimum, or if one of the other local optima were selected for any given run. The graph of the function that was explored for all of these runs can be seen in Figure 3.1

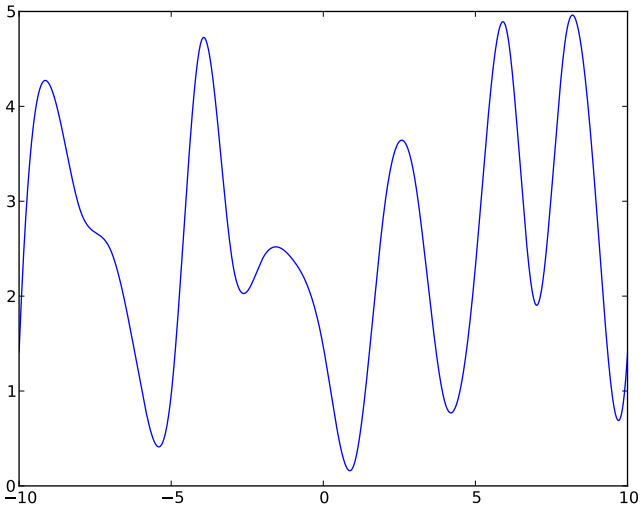

Figure 3.1: The graph of the interpolated function that was explored in the algorithm in Appendix A. The x-axis is the solution space, and the y-axis is the corresponding fitness for any given part of the solution space. Note the global maximum at approximately 8.

3.2 Results of Simulation

3.2.1 1st Simulation Run

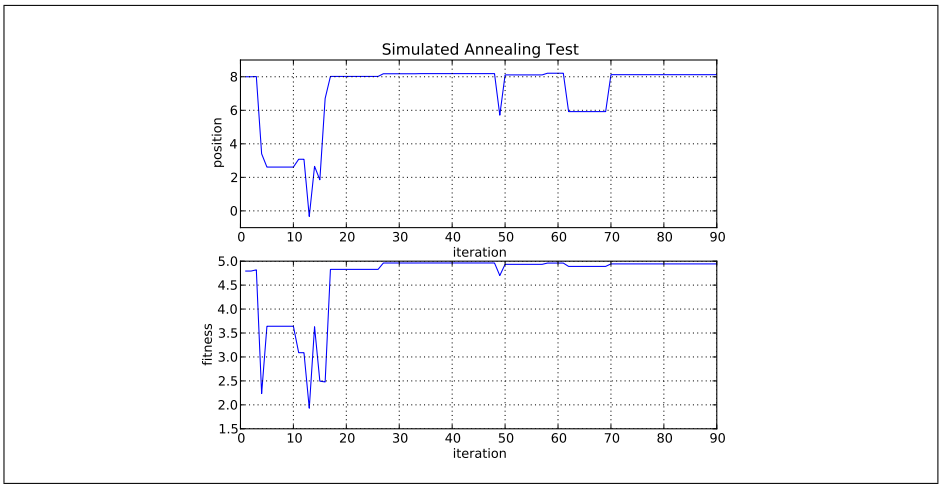

Figure 3.2: Results of 1st Experiment

The Temperature was: 10

The Boltzmann Constant was: 0.245

The Interval Weight was: 5.0

The Neighborhood Selection Method Chosen was: Constant

The Annealing Schedule was: Exponential

The relevant Annealing Constant was: 0.95

The probability method chosen was: Metropolis

The termination temperature set was: 0.1

The cost function chosen was: Random Interpolation

The optimal input found was: 8.12453669701

The maximum value found was: 4.94273200262

The optima reached was: Global

3.2.2 2nd Simulation Run

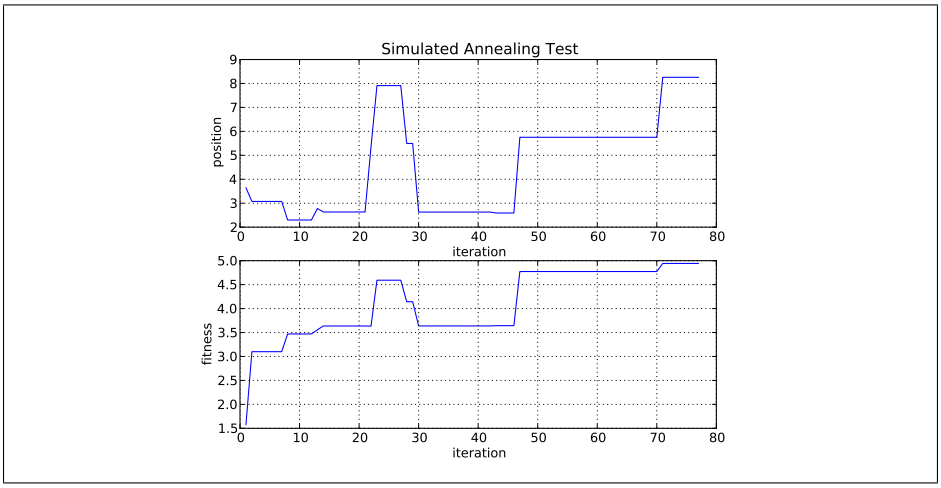

Figure 3.3: Results of 2nd Experiment

The Temperature was: 5

The Boltzmann Constant was: 0.245

The Interval Weight was: 5.0

The Neighborhood Selection Method Chosen was: Constant

The Annealing Schedule was: Exponential

The relevant Annealing Constant was: 0.95

The probability method chosen was: Metropolis

The termination temperature set was: 0.1

The cost function chosen was: Random Interpolation

The optimal input found was: 8.25702992337

The maximum value found was: 4.94262557569

The optima reached was: Global

3.2.3 3rd Simulation Run

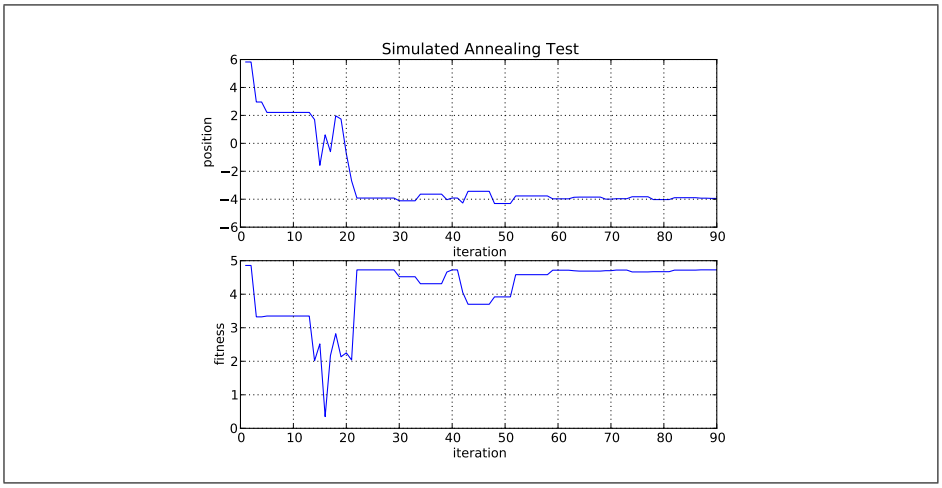

Figure 3.4: Results of 3rd Experiment

The Temperature was: 10

The Boltzmann Constant was: 0.245

The Interval Weight was: 0.4

The Neighborhood Selection Method Chosen was: Square Root

The Annealing Schedule was: Exponential

The relevant Annealing Constant was: 0.95

The probability method chosen was: Metropolis

The termination temperature set was: 0.1

The cost function chosen was: Random Interpolation

The optimal input found was: -3.94145196297

The maximum value found was: 4.72607762918

The optima reached was: Local

3.2.4 4th Simulation Run

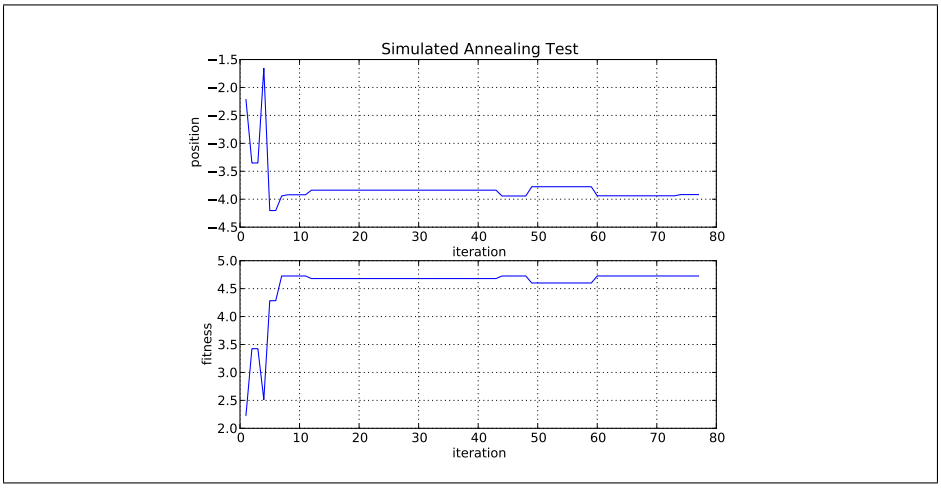

Figure 3.5: Results of 4th Experiment

The Temperature was: 5

The Boltzmann Constant was: 0.245

The Interval Weight was: 0.2

The Neighborhood Selection Method Chosen was: Square Root

The Annealing Schedule was: Exponential

The relevant Annealing Constant was: 0.95

The probability method chosen was: Metropolis

The termination temperature set was: 0.1

The cost function chosen was: Random Interpolation

The optimal input found was: -3.91731183007

The maximum value found was: 4.72582418859

The optima reached was: Local

3.2.5 5th Simulation Run

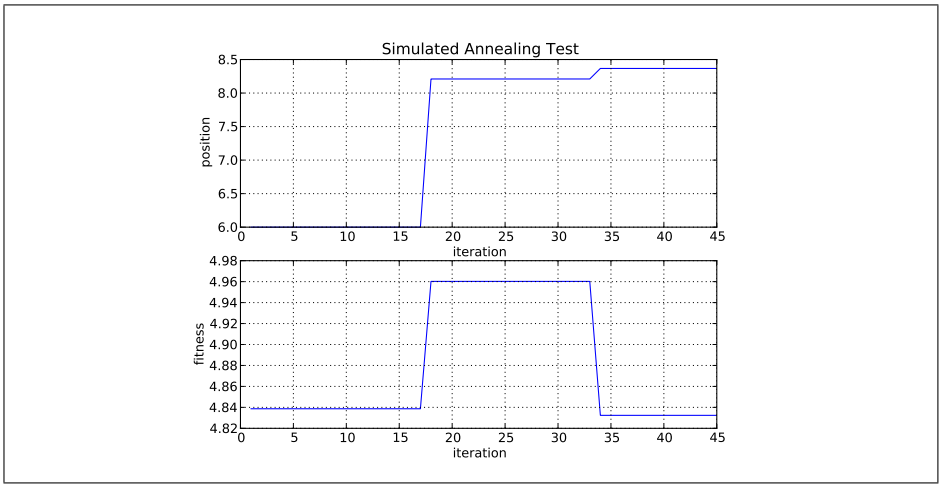

Figure 3.6: Results of 5th Experiment

The Temperature was: 1

The Boltzmann Constant was: 0.245

The Interval Weight was: 0.04

The Neighborhood Selection Method Chosen was: Square Root

The Annealing Schedule was: Exponential

The relevant Annealing Constant was: 0.95

The probability method chosen was: Metropolis

The termination temperature set was: 0.1

The cost function chosen was: Random Interpolation

The optimal input found was: 8.36686935583

The maximum value found was: 4.83238399159

The optima reached was: Global

3.2.6 6th Simulation Run

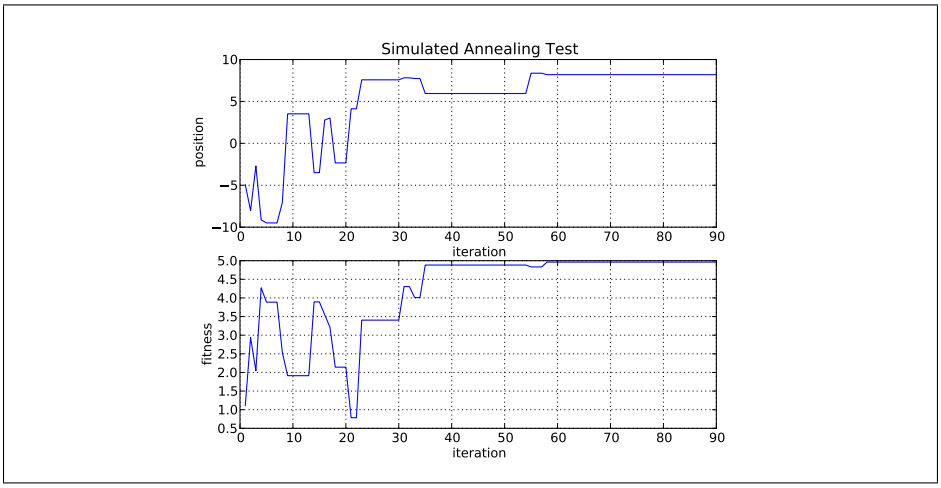

Figure 3.7: Results of 6th Experiment

The Temperature was: 10

The Boltzmann Constant was: 0.245

The Interval Weight was: 0.04

The Neighborhood Selection Method Chosen was: Square Root

The Annealing Schedule was: Exponential

The relevant Annealing Constant was: 0.95

The probability method chosen was: Alternate

The termination temperature set was: 0.1

The cost function chosen was: Random Interpolation

The optimal input found was: 8.19310048306

The maximum value found was: 4.96191639881

The optima reached was: Global

3.2.7 7th Simulation Run

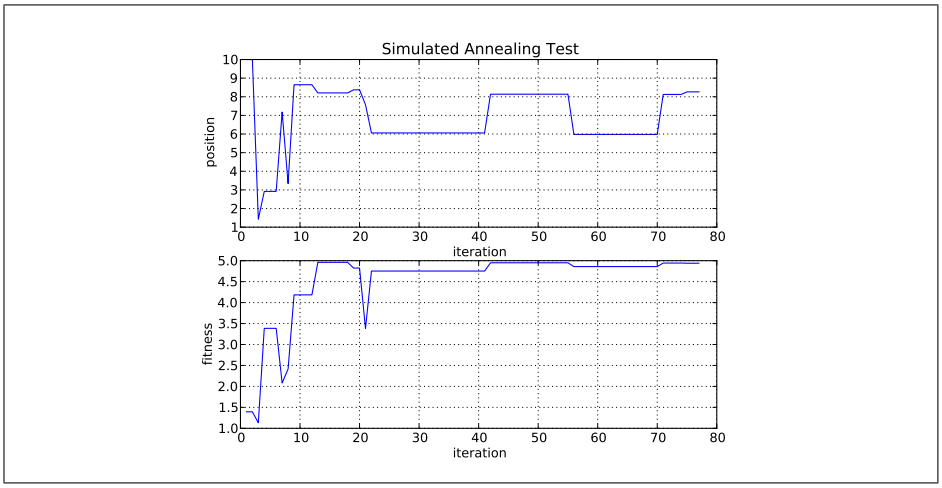

Figure 3.8: Results of 7th Experiment

The Temperature was: 5

The Boltzmann Constant was: 0.245

The Interval Weight was: 0.02

The Neighborhood Selection Method Chosen was: Square Root

The Annealing Schedule was: Exponential

The relevant Annealing Constant was: 0.95

The probability method chosen was: Alternate

The termination temperature set was: 0.1

The cost function chosen was: Random Interpolation

The optimal input found was: 8.26344538045

The maximum value found was: 4.93879936195

The optima reached was: Global

3.2.8 8th Simulation Run

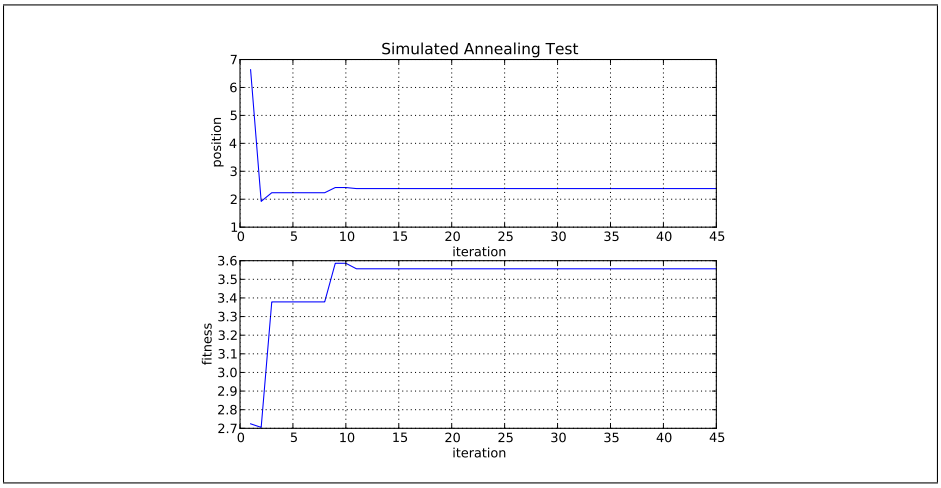

Figure 3.9: Results of 8th Experiment

The Temperature was: 1

The Boltzmann Constant was: 0.245

The Interval Weight was: 0.04

The Neighborhood Selection Method Chosen was: Square Root

The Annealing Schedule was: Exponential

The relevant Annealing Constant was: 0.95

The probability method chosen was: Alternate

The termination temperature set was: 0.1

The cost function chosen was: Random Interpolation

The optimal input found was: 2.38064428537

The maximum value found was: 3.55647132056

The optima reached was: Local

3.2.9 9th Simulation Run

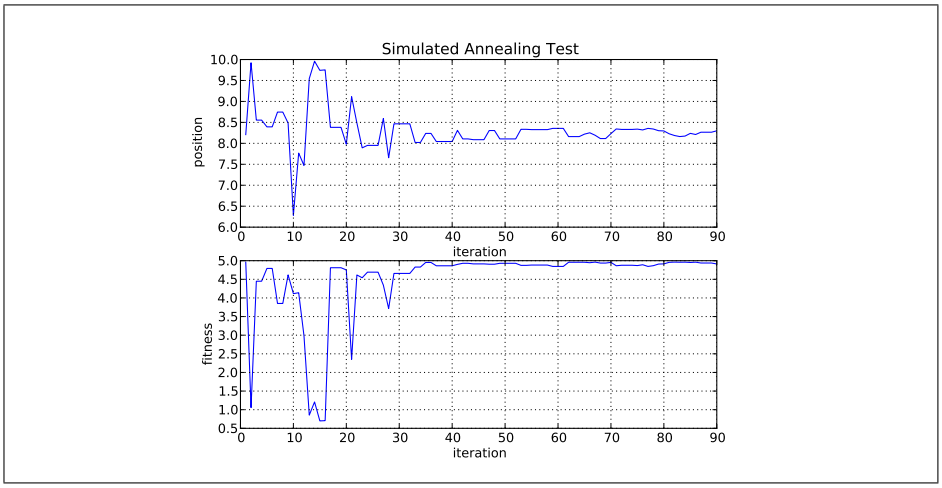

Figure 3.10: Results of 9th Experiment

The Temperature was: 10

The Boltzmann Constant was: 0.245

The Interval Weight was: 0.5

The Neighborhood Selection Method Chosen was: Linear

The Annealing Schedule was: Exponential

The relevant Annealing Constant was: 0.95

The probability method chosen was: Metropolis

The termination temperature set was: 0.1

The cost function chosen was: Random Interpolation

The optimal input found was: 8.29950940596

The maximum value found was: 4.91110900971

The optima reached was: Global

3.2.10 10th Simulation Run

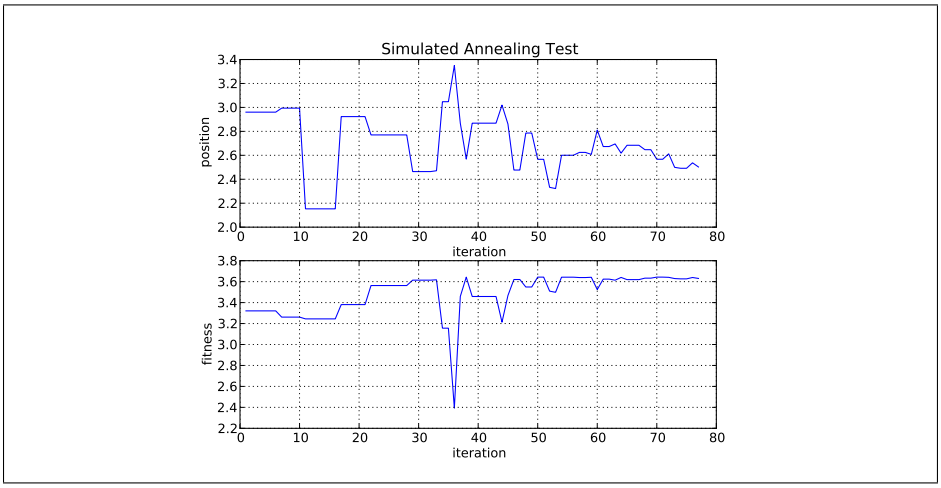

Figure 3.11: Results of 10th Experiment

The Temperature was: 5

The Boltzmann Constant was: 0.245

The Interval Weight was: 1.0

The Neighborhood Selection Method Chosen was: Linear

The Annealing Schedule was: Exponential

The relevant Annealing Constant was: 0.95

The probability method chosen was: Metropolis

The termination temperature set was: 0.1

The cost function chosen was: Random Interpolation

The optimal input found was: 2.50249440476

The maximum value found was: 3.6303864234

The optima reached was: Local

3.2.11 11th Simulation Run

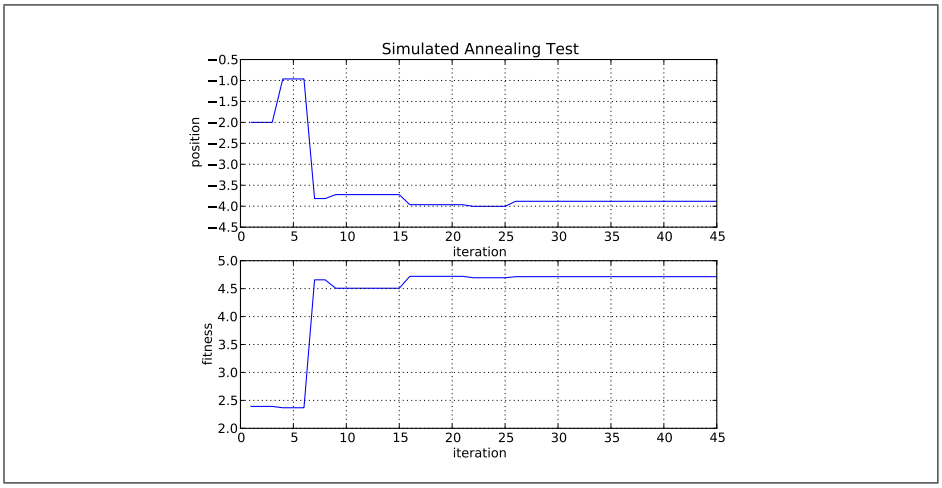

Figure 3.12: Results of 11th Experiment

The Temperature was: 1

The Boltzmann Constant was: 0.245

The Interval Weight was: 5.0

The Neighborhood Selection Method Chosen was: Linear

The Annealing Schedule was: Exponential

The relevant Annealing Constant was: 0.95

The probability method chosen was: Metropolis

The termination temperature set was: 0.1

The cost function chosen was: Random Interpolation

The optimal input found was: -3.88278730133

The maximum value found was: 4.71396455154

The optima reached was: Local

3.2.12 12th Simulation Run

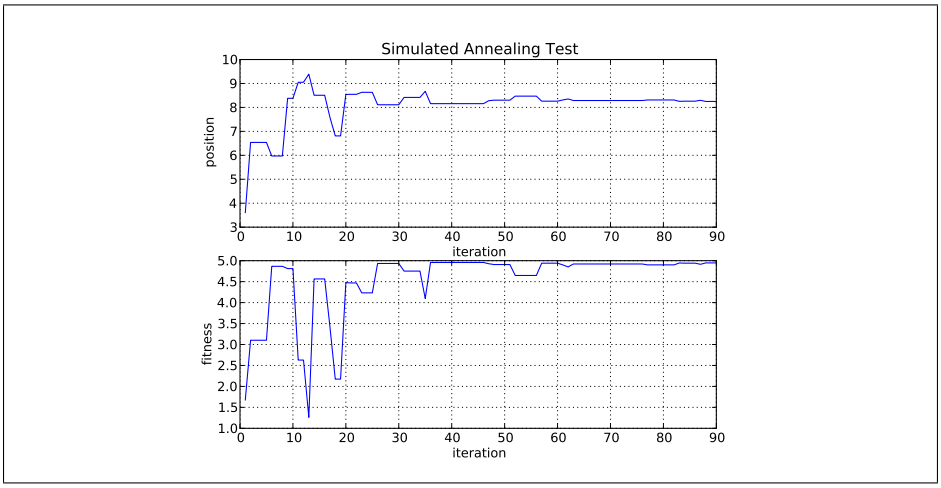

Figure 3.13: Results of 12th Experiment

The Temperature was: 10

The Boltzmann Constant was: 0.245

The Interval Weight was: 0.5

The Neighborhood Selection Method Chosen was: Linear

The Annealing Schedule was: Exponential

The relevant Annealing Constant was: 0.95

The probability method chosen was: Alternate

The termination temperature set was: 0.1

The cost function chosen was: Random Interpolation

The optimal input found was: 8.24916033885

The maximum value found was: 4.9468572584

The optima reached was: Global

3.2.13 13th Simulation Run

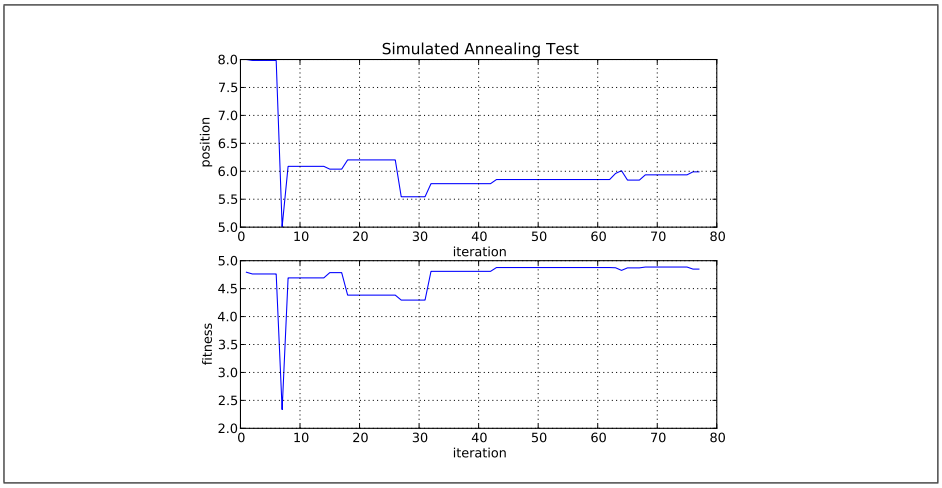

Figure 3.14: Results of 13th Experiment

The Temperature was: 5

The Boltzmann Constant was: 0.245

The Interval Weight was: 1.0

The Neighborhood Selection Method Chosen was: Linear

The Annealing Schedule was: Exponential

The relevant Annealing Constant was: 0.95

The probability method chosen was: Alternate

The termination temperature set was: 0.1

The cost function chosen was: Random Interpolation

The optimal input found was: 5.99006003943

The maximum value found was: 4.84934812591

The optima reached was: Local

3.2.14 14th Simulation Run

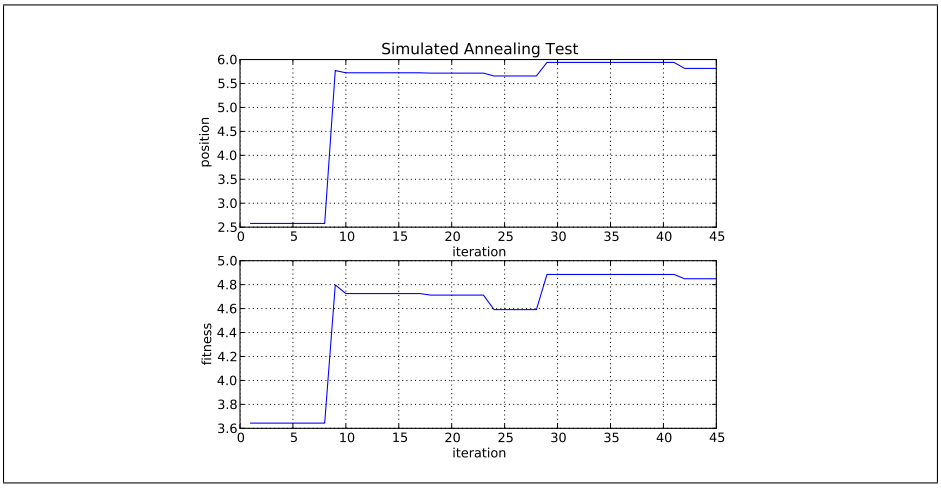

Figure 3.15: Results of 14th Experiment

The Temperature was: 1

The Boltzmann Constant was: 0.245

The Interval Weight was: 5.0

The Neighborhood Selection Method Chosen was: Linear

The Annealing Schedule was: Exponential

The relevant Annealing Constant was: 0.95

The probability method chosen was: Alternate

The termination temperature set was: 0.1

The cost function chosen was: Random Interpolation

The optimal input found was: 5.81446248631

The maximum value found was: 4.84908964862

The optima reached was: Local

3.2.15 15th Simulation Run

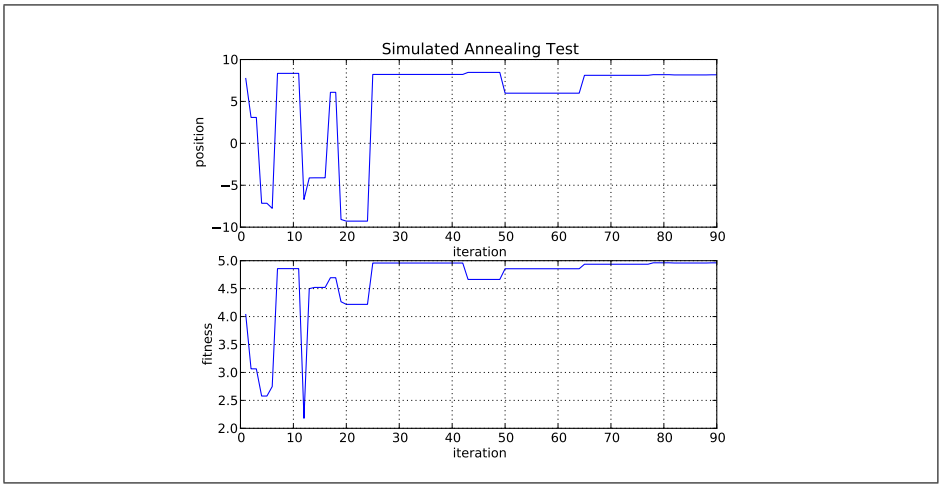

Figure 3.16: Results of 15th Experiment

The Temperature was: 10

The Boltzmann Constant was: 0.245

The Interval Weight was: 0.05

The Neighborhood Selection Method Chosen was: Squared

The Annealing Schedule was: Exponential

The relevant Annealing Constant was: 0.95

The probability method chosen was: Metropolis

The termination temperature set was: 0.1

The cost function chosen was: Random Interpolation

The optimal input found was: 8.17734638299

The maximum value found was: 4.96125587134

The optima reached was: Global

3.2.16 16th Simulation Run

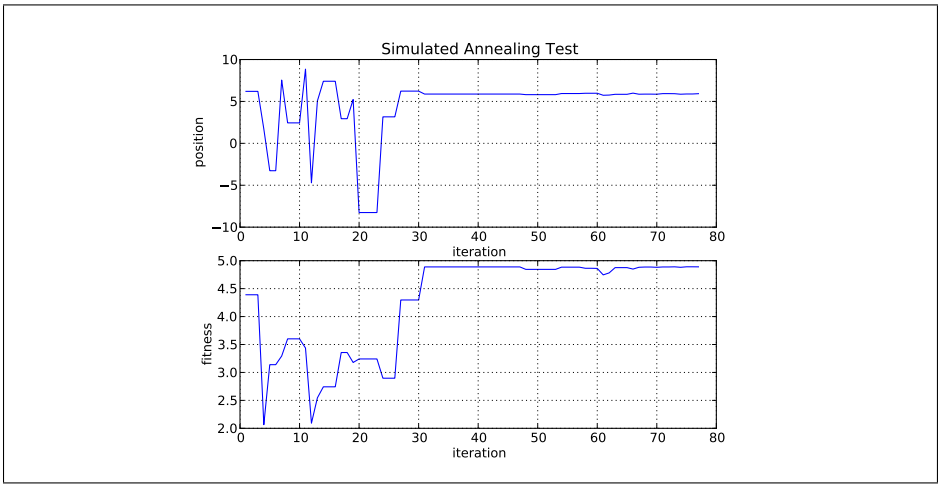

Figure 3.17: Results of 16th Experiment

The Temperature was: 5

The Boltzmann Constant was: 0.245

The Interval Weight was: 0.2

The Neighborhood Selection Method Chosen was: Squared

The Annealing Schedule was: Exponential

The relevant Annealing Constant was: 0.95

The probability method chosen was: Metropolis

The termination temperature set was: 0.1

The cost function chosen was: Random Interpolation

The optimal input found was: 5.92454497566

The maximum value found was: 4.88964836538

The optima reached was: Local

3.2.17 17th Simulation Run

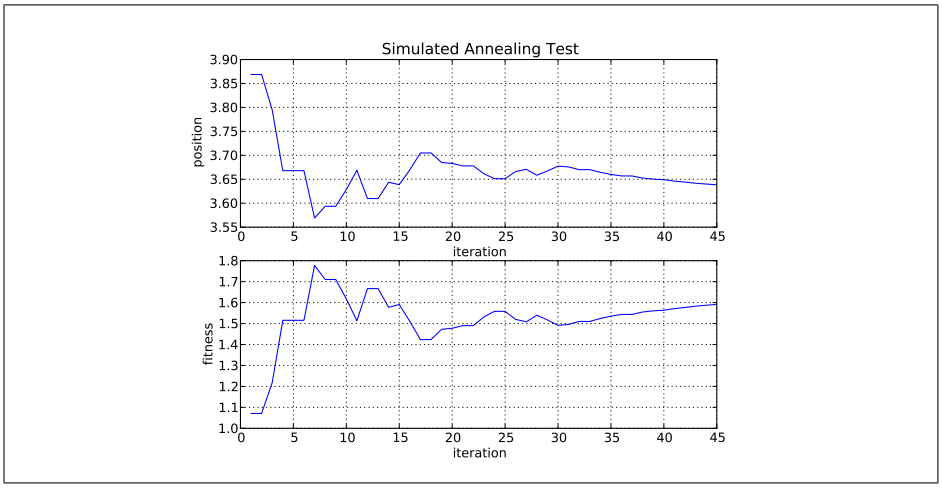

Figure 3.18: Results of 17th Experiment

The Temperature was: 1

The Boltzmann Constant was: 0.245

The Interval Weight was: 5.0

The Neighborhood Selection Method Chosen was: Squared

The Annealing Schedule was: Exponential

The relevant Annealing Constant was: 0.95

The probability method chosen was: Metropolis

The termination temperature set was: 0.1

The cost function chosen was: Random Interpolation

The optimal input found was: 3.63840734657

The maximum value found was: 1.59143401283

The optima reached was: Nonoptimal

3.2.18 18th Simulation Run

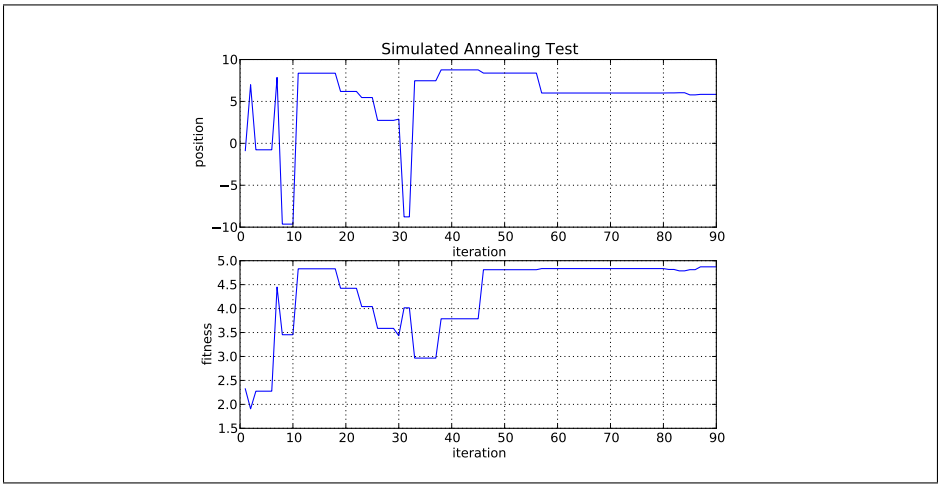

Figure 3.19: Results of 18th Experiment

The Temperature was: 10

The Boltzmann Constant was: 0.245

The Interval Weight was: 0.05

The Neighborhood Selection Method Chosen was: Squared

The Annealing Schedule was: Exponential

The relevant Annealing Constant was: 0.95

The probability method chosen was: Alternate

The termination temperature set was: 0.1

The cost function chosen was: Random Interpolation

The optimal input found was: 5.84291572758

The maximum value found was: 4.87167478904

The optima reached was: Local

3.2.19 19th Simulation Run

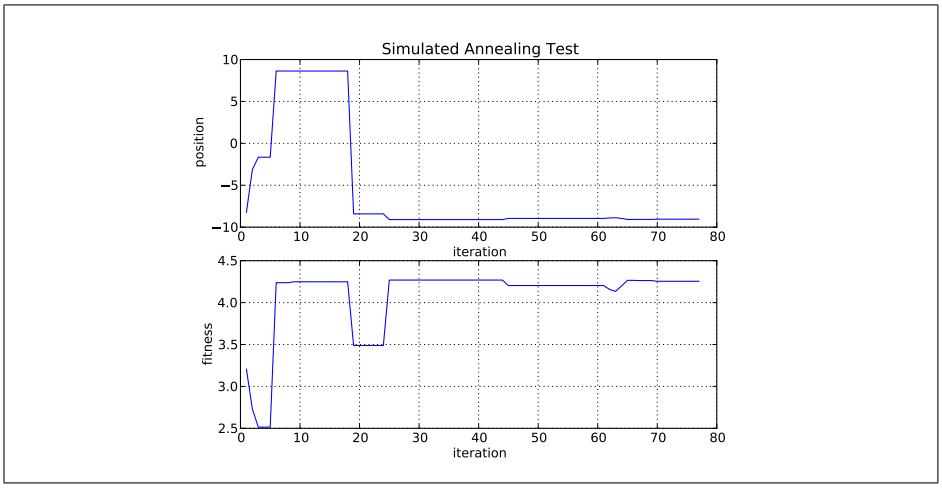

Figure 3.20: Results of 19th Experiment

The Temperature was: 5

The Boltzmann Constant was: 0.245

The Interval Weight was: 0.2

The Neighborhood Selection Method Chosen was: Squared

The Annealing Schedule was: Exponential

The relevant Annealing Constant was: 0.95

The probability method chosen was: Alternate

The termination temperature set was: 0.1

The cost function chosen was: Random Interpolation

The optimal input found was: -9.04789162475

The maximum value found was: 4.25423352425

The optima reached was: Local

3.2.20 20th Simulation Run

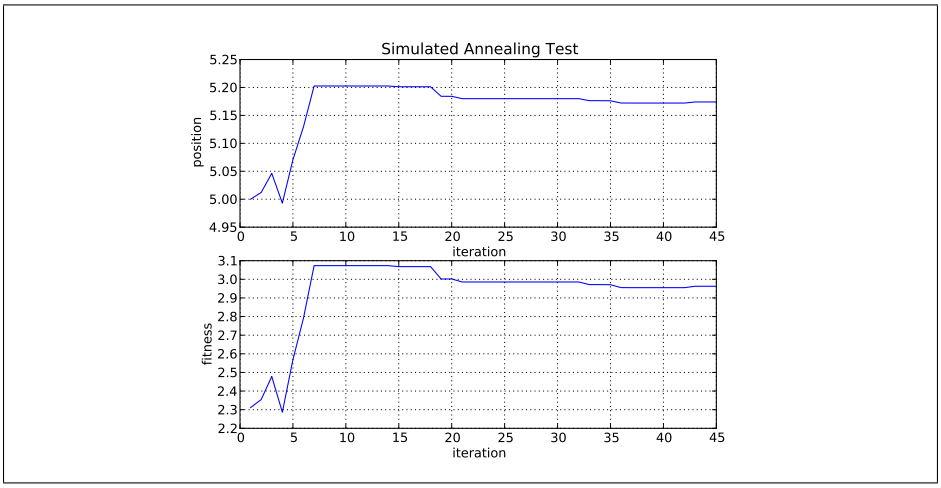

Figure 3.21: Results of 20th Experiment

The Temperature was: 1

The Boltzmann Constant was: 0.245

The Interval Weight was: 5.0

The Neighborhood Selection Method Chosen was: Squared

The Annealing Schedule was: Exponential

The relevant Annealing Constant was: 0.95

The probability method chosen was: Alternate

The termination temperature set was: 0.1

The cost function chosen was: Random Interpolation

The optimal input found was: 5.17406360604

The maximum value found was: 2.96278314043

The optima reached was: Local

3.2.21 21st Simulation Run

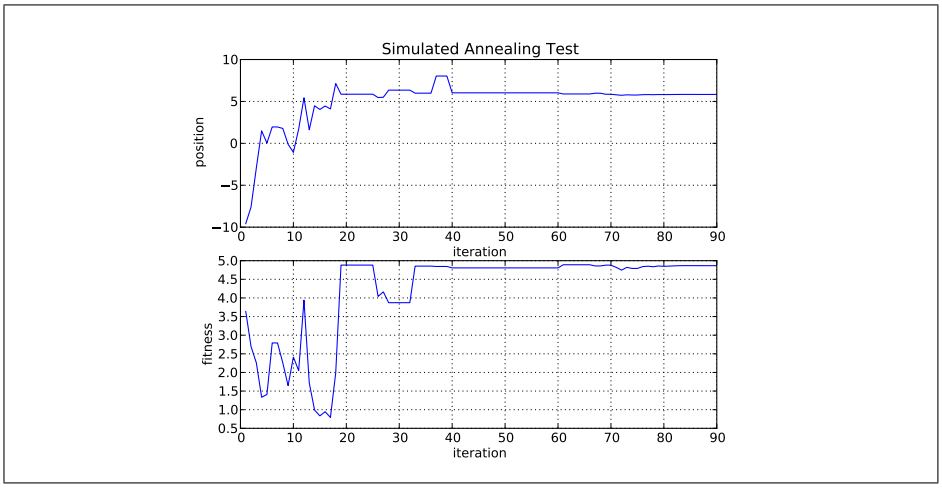

Figure 3.22: Results of 21st Experiment

The Temperature was: 10

The Boltzmann Constant was: 0.245

The Interval Weight was: 5.5

The Neighborhood Selection Method Chosen was: Exponential

The Annealing Schedule was: Exponential

The relevant Annealing Constant was: 0.95

The probability method chosen was: Metropolis

The termination temperature set was: 0.1

The cost function chosen was: Random Interpolation

The optimal input found was: 5.8349624473

The maximum value found was: 4.8661724921

The optima reached was: Local

3.2.22 22nd Simulation Run

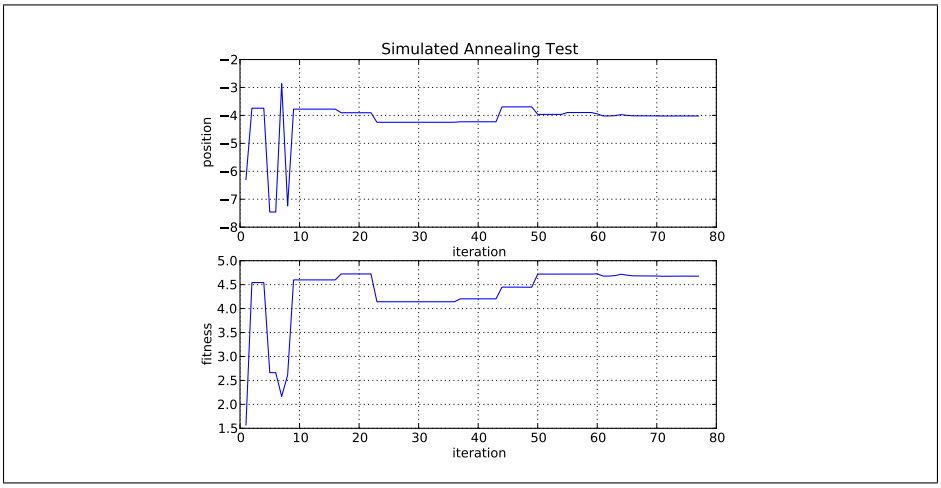

Figure 3.23: Results of 22nd Experiment

The Temperature was: 5

The Boltzmann Constant was: 0.245

The Interval Weight was: 6.1

The Neighborhood Selection Method Chosen was: Exponential

The Annealing Schedule was: Exponential

The relevant Annealing Constant was: 0.95

The probability method chosen was: Metropolis

The termination temperature set was: 0.1

The cost function chosen was: Random Interpolation

The optimal input found was: -4.02034824486

The maximum value found was: 4.67704142695

The optima reached was: Local

3.2.23 23rd Simulation Run

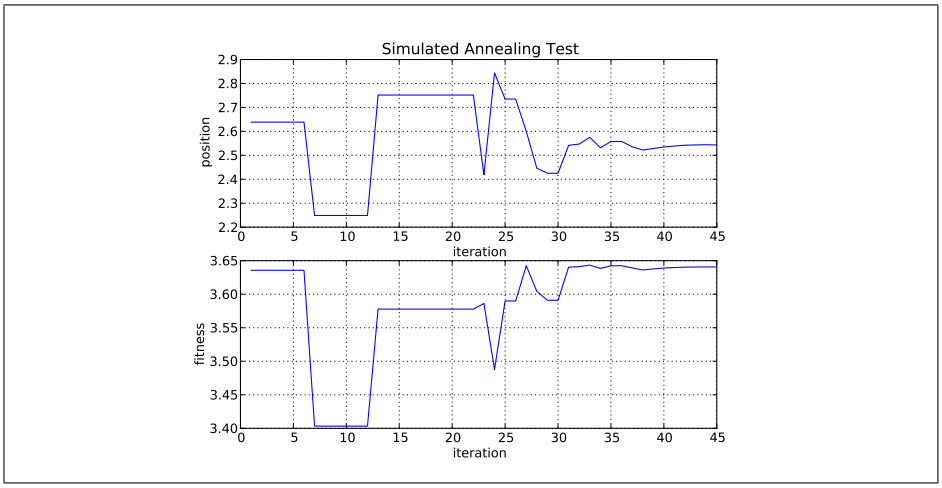

Figure 3.24: Results of 23rd Experiment

The Temperature was: 1

The Boltzmann Constant was: 0.245

The Interval Weight was: 13.6

The Neighborhood Selection Method Chosen was: Exponential

The Annealing Schedule was: Exponential

The relevant Annealing Constant was: 0.95

The probability method chosen was: Metropolis

The termination temperature set was: 0.1

The cost function chosen was: Random Interpolation

The optimal input found was: 2.54326998373

The maximum value found was: 3.64055218711

The optima reached was: Local

3.2.24 24th Simulation Run

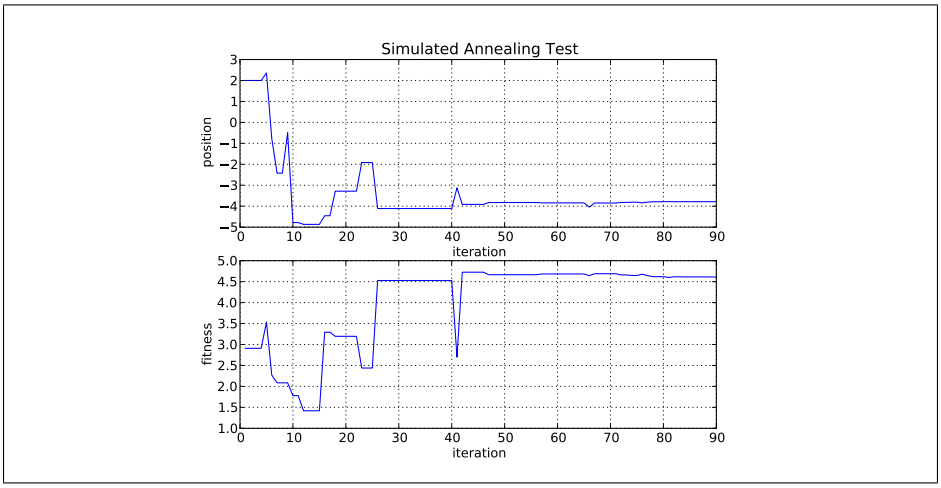

Figure 3.25: Results of 24th Experiment

The Temperature was: 10

The Boltzmann Constant was: 0.245

The Interval Weight was: 5.5

The Neighborhood Selection Method Chosen was: Exponential

The Annealing Schedule was: Exponential

The relevant Annealing Constant was: 0.95

The probability method chosen was: Alternate

The termination temperature set was: 0.1

The cost function chosen was: Random Interpolation

The optimal input found was: -3.78353911027

The maximum value found was: 4.61158829596

The optima reached was: Local

3.2.25 25th Simulation Run

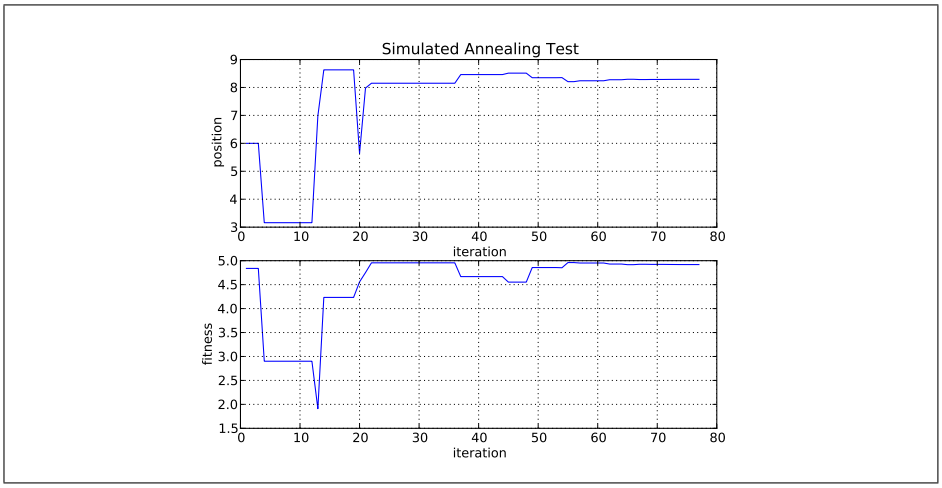

Figure 3.26: Results of 25th Experiment

The Temperature was: 5

The Boltzmann Constant was: 0.245

The Interval Weight was: 6.1

The Neighborhood Selection Method Chosen was: Exponential

The Annealing Schedule was: Exponential

The relevant Annealing Constant was: 0.95

The probability method chosen was: Alternate

The termination temperature set was: 0.1

The cost function chosen was: Random Interpolation

The optimal input found was: 8.290748868

The maximum value found was: 4.91878894657

The optima reached was: Global

3.2.26 26th Simulation Run

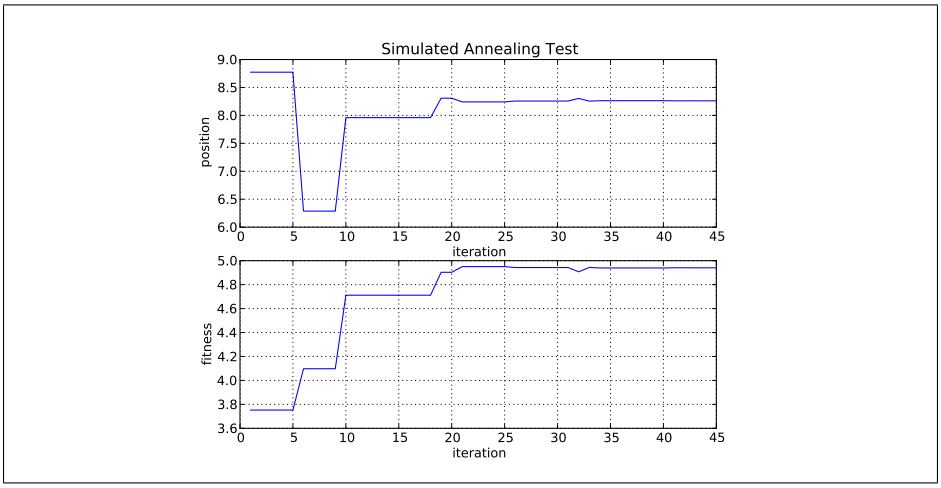

Figure 3.27: Results of 26th Experiment

The Temperature was: 1

The Boltzmann Constant was: 0.245

The Interval Weight was: 13.6

The Neighborhood Selection Method Chosen was: Exponential

The Annealing Schedule was: Exponential

The relevant Annealing Constant was: 0.95

The probability method chosen was: Alternate

The termination temperature set was: 0.1

The cost function chosen was: Random Interpolation

The optimal input found was: 8.26080563276

The maximum value found was: 4.94041449576

The optima reached was: Global

3.2.27 27th Simulation Run

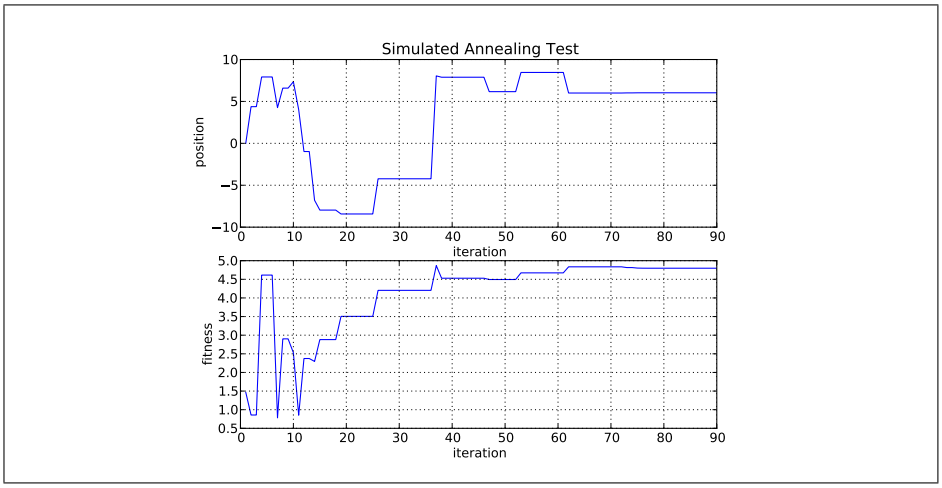

Figure 3.28: Results of 27th Experiment

The Temperature was: 10

The Boltzmann Constant was: 0.245

The Interval Weight was: 127.0

The Neighborhood Selection Method Chosen was: Normal Distribution

The Annealing Schedule was: Exponential

The relevant Annealing Constant was: 0.95

The probability method chosen was: Alternate

The termination temperature set was: 0.1

The cost function chosen was: Random Interpolation

The optimal input found was: 6.03075787558

The maximum value found was: 4.7978270401

The optima reached was: Local

3.2.28 28th Simulation Run

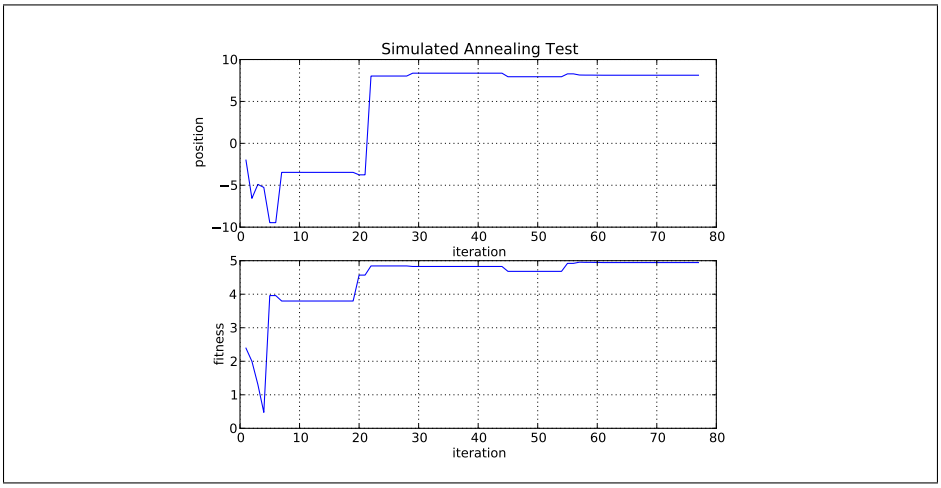

Figure 3.29: Results of 28th Experiment

The Temperature was: 5

The Boltzmann Constant was: 0.245

The Interval Weight was: 63.6

The Neighborhood Selection Method Chosen was: Normal Distribution

The Annealing Schedule was: Exponential

The relevant Annealing Constant was: 0.95

The probability method chosen was: Alternate

The termination temperature set was: 0.1

The cost function chosen was: Random Interpolation

The optimal input found was: 8.12612264053

The maximum value found was: 4.94366208577

The optima reached was: Global

3.2.29 29th Simulation Run

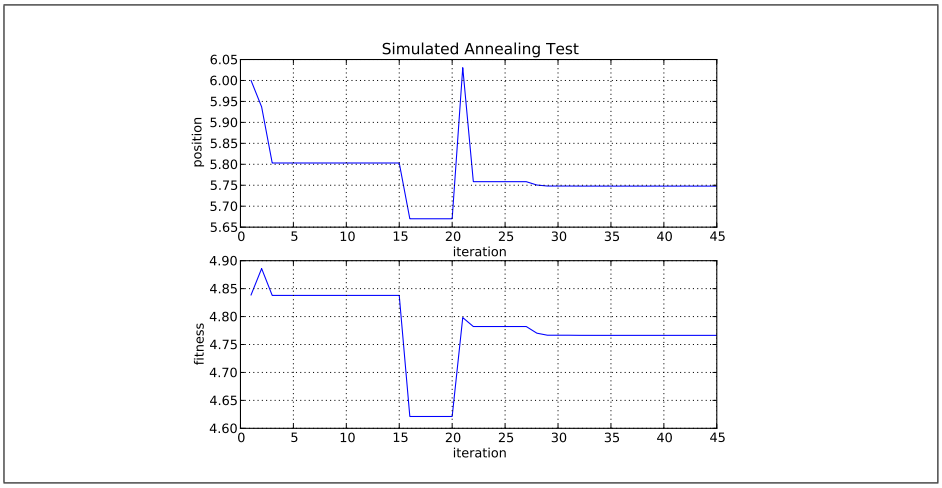

Figure 3.30: Results of 29th Experiment

The Temperature was: 1

The Boltzmann Constant was: 0.245

The Interval Weight was: 20.7

The Neighborhood Selection Method Chosen was: Normal Distribution

The Annealing Schedule was: Exponential

The relevant Annealing Constant was: 0.95

The probability method chosen was: Alternate

The termination temperature set was: 0.1

The cost function chosen was: Random Interpolation

The optimal input found was: 5.74795311155

The maximum value found was: 4.76635076685

The optima reached was: Local

3.2.30 30th Simulation Run

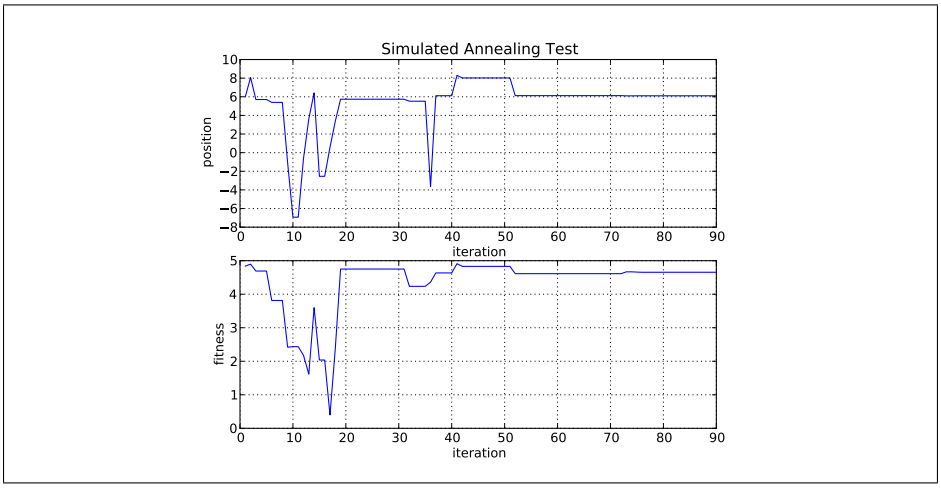

Figure 3.31: Results of 30th Experiment

The Temperature was: 10

The Boltzmann Constant was: 0.245

The Interval Weight was: 127.0

The Neighborhood Selection Method Chosen was: Normal Distribution

The Annealing Schedule was: Exponential

The relevant Annealing Constant was: 0.95

The probability method chosen was: Metropolis

The termination temperature set was: 0.1

The cost function chosen was: Random Interpolation

The optimal input found was: 6.10503840733

The maximum value found was: 4.65468377356

The optima reached was: Local

3.2.31 31st Simulation Run

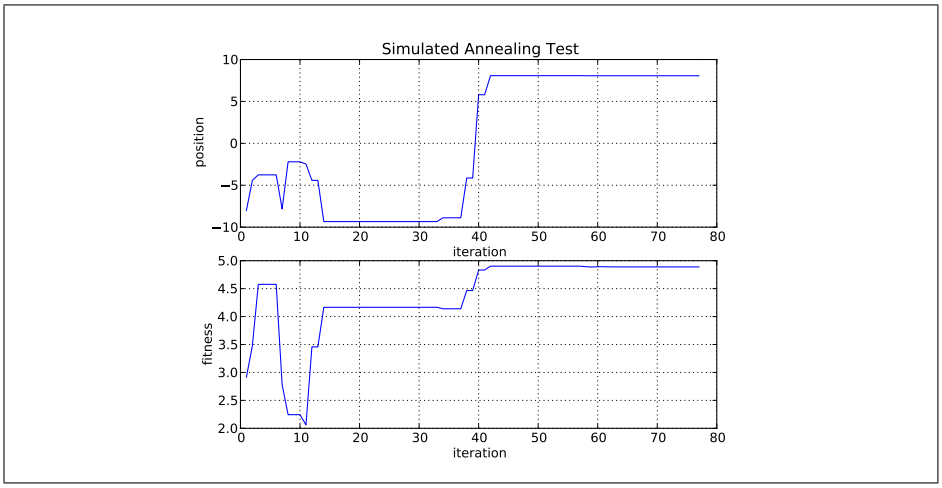

Figure 3.32: Results of 31st Experiment

The Temperature was: 5

The Boltzmann Constant was: 0.245

The Interval Weight was: 63.6

The Neighborhood Selection Method Chosen was: Normal Distribution

The Annealing Schedule was: Exponential

The relevant Annealing Constant was: 0.95

The probability method chosen was: Metropolis

The termination temperature set was: 0.1

The cost function chosen was: Random Interpolation

The optimal input found was: 8.06337249662

The maximum value found was: 4.8885707594

The optima reached was: Global

3.2.32 32nd Simulation Run

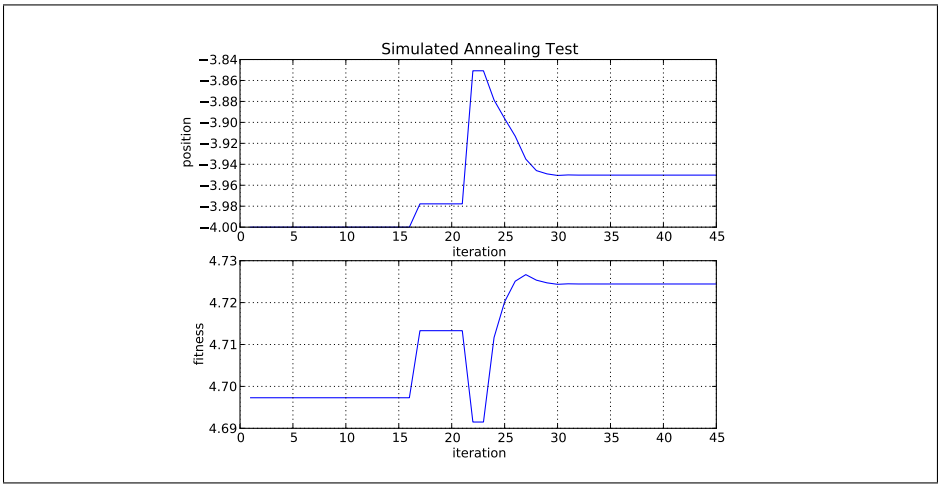

Figure 3.33: Results of 32nd Experiment

The Temperature was: 1

The Boltzmann Constant was: 0.245

The Interval Weight was: 20.7

The Neighborhood Selection Method Chosen was: Normal Distribution

The Annealing Schedule was: Exponential

The relevant Annealing Constant was: 0.95

The probability method chosen was: Metropolis

The termination temperature set was: 0.1

The cost function chosen was: Random Interpolation

The optimal input found was: -3.95033095606

The maximum value found was: 4.72443970661

The optima reached was: Local

3.3 General Remarks on Simulation Results

The only thing that needs to be determined for the analysis of these simulation results is whether or not the run converges on the global maximum, on one of the local maxima, or if it did not converge at all. Of these, only the 17th run of the simulation did not converge to an optimal solution. This suggests that for the given conditions of that particular run that convergence to an optimal solution is unlikely.

For the rest, it seems that for higher temperatures that the algorithm is likely to converge on the global optimum independent of the other choices that are made. This suggests that temperature is the most important factor, along with the termination condition, in determining whether or not the algorithm will converge on the global maximum.

In addition, it was determined that the selection method for determining the neighborhood that grew as a function of temperature (called the Normal Distribution method in the algorithm) converged to the global maximum the least often. This motivated the abandoning of this neighborhood selection method in the physical experiment algorithm.

3.4 Results of Physical Experiment

The algorithm that was written in Appendix B, with the corresponding module for interaction with the Arduino defined in Appendix C, was used to obtain one set of results for the physical experiment. The graphs showing the selection of the method are shown after the logfile. The logfile from this experiment is:

These were the selections made for this experiment:

The Neighborhood Selection Method Chosen was: Square Root

The probability method chosen was: Metropolis

The number of pulses per pulse train was: 5

The number of experiments ran was: 1

The number of points of data the Arduino collected was: 20

For the 1st experiment, the optimal timing for the 1st element of the sequence is: 1508

The corresponding maximum voltage attained for this is: 0.9814453125

For the 1st experiment, the optimal timing for the 2nd element of the sequence is: 3983

The corresponding maximum voltage attained for this is: 0.9716796875

For the 1st experiment, the optimal timing for the 3rd element of the sequence is: 2254

The corresponding maximum voltage attained for this is: 0.83984375

For the 1st experiment, the optimal timing for the 4th element of the sequence is: 3984

The corresponding maximum voltage attained for this is: 0.8642578125

For the 1st experiment, the optimal timing for the 5th element of the sequence is: 691

The corresponding maximum voltage attained for this is: 0.8837890625

For the 1st experiment, the optimal pulse train was: [1508, 3983, 2254, 3984, 691]

The corresponding fitness for this pulse train was: 0.8837890625

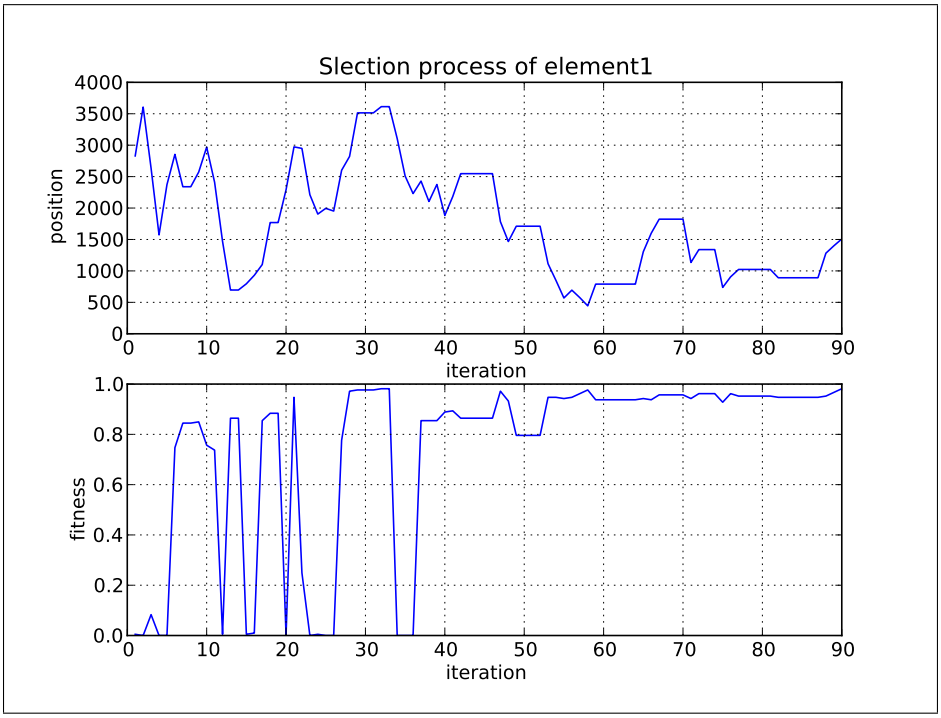

Figure 3.34: 1st pulse timing selection

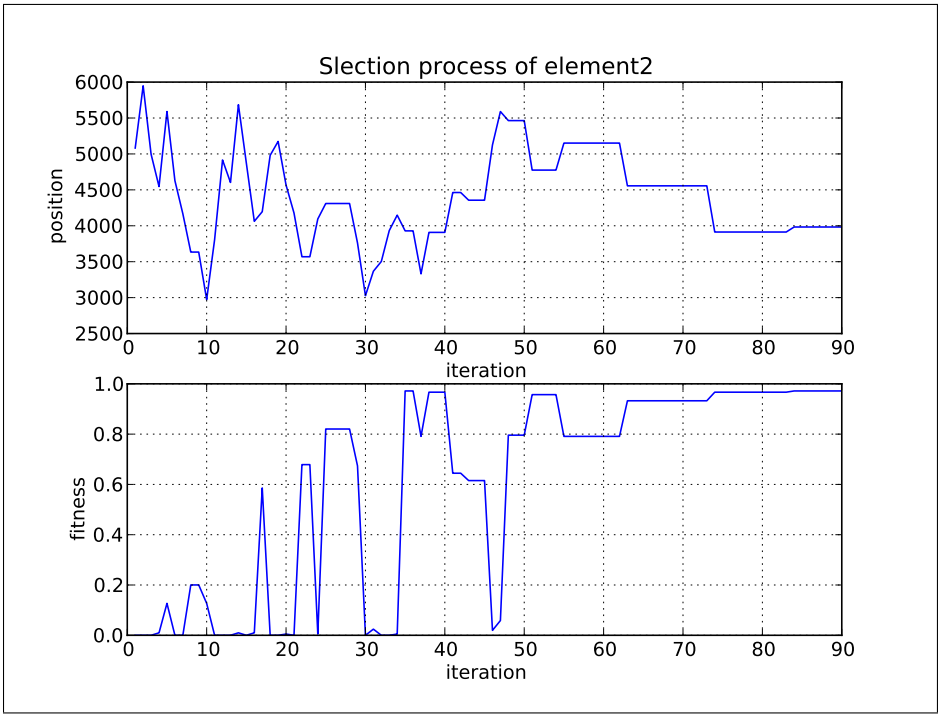

Figure 3.35: 2nd pulse timing selection

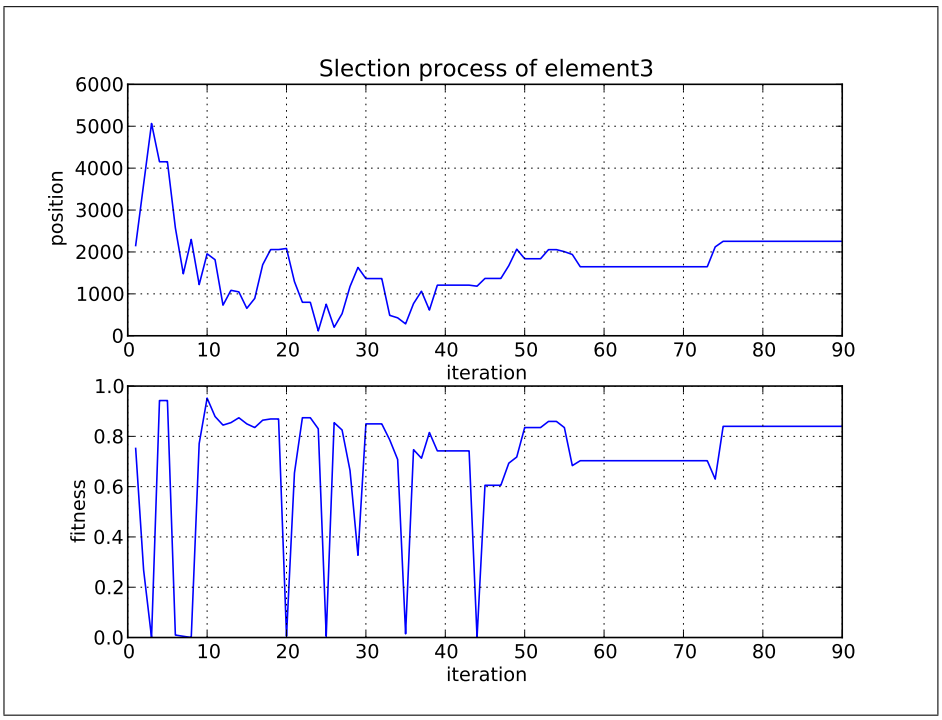

Figure 3.36: 3rd pulse timing selection

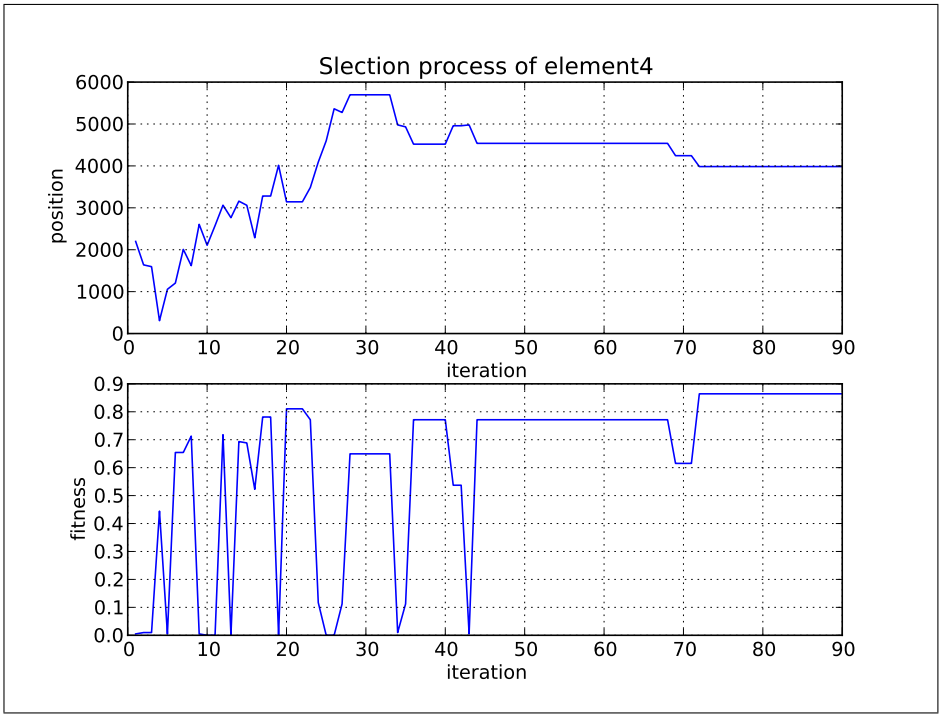

Figure 3.37: 4th pulse timing selection

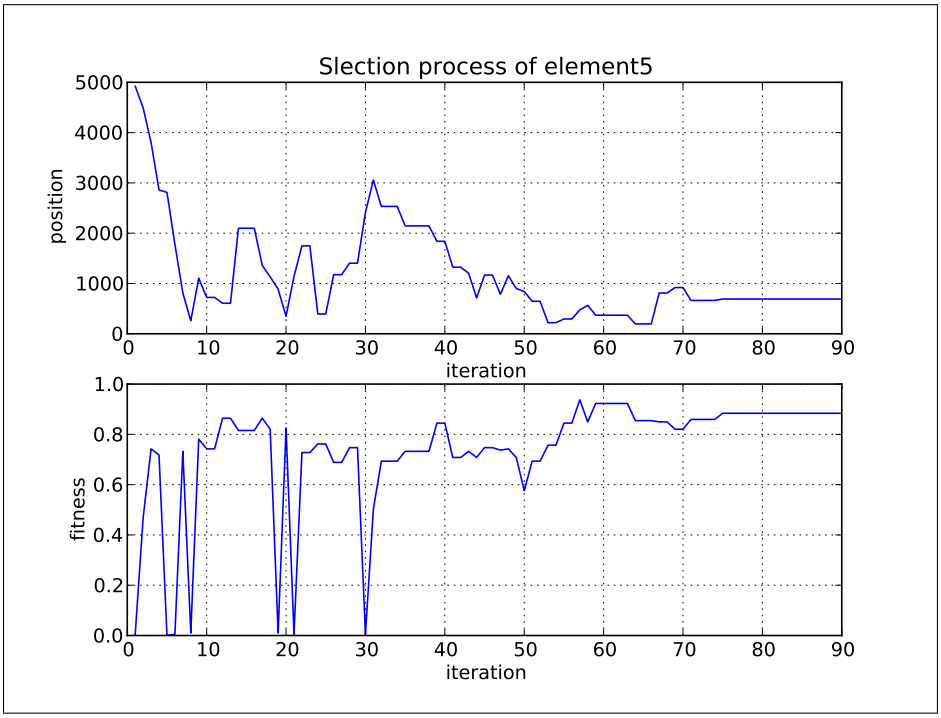

Figure 3.38: 5th pulse timing selection

3.5 Analysis of Physical Experiment

For this experiment, the voltage was taken across the resistor. This means that the voltage as a function of time is related to the first derivative of charge, which means that the voltage behaves as a sine function allowing for our assumptions about when the optimal time occurrs to be true. In addition, it can also be assumed that there is no significant loss of charge from when the last pulse was delivered to the system and when the measurement of the amplitude was taken. Therefore, the charge after the previous pulse is assumed to be the charge present when the voltage measurement was taken. We can use the established equations in the Motivation section (specifically 1.35 and 1.36) in order to determine what the charge is for that moment in time. From there we can obtain the electric potential as a function of charge, and this can be used to find the potential energy function for the LRC.

Using the Mathematica code in Appendix E with the results from this experiment, a fit for the electric potential is obtained in Figure 3.39.

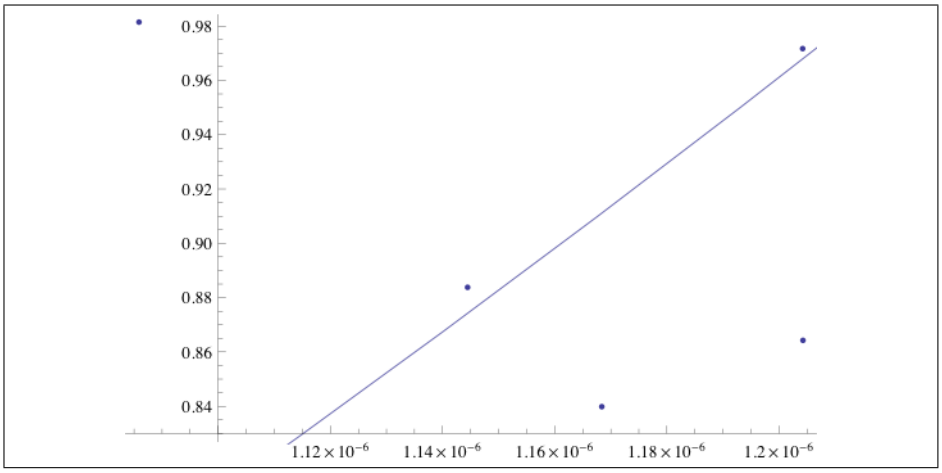

Figure 3.39: Electric potential fit

Chapter 4

Discussion

4.1 Conclusions

The experiment did perform within expectations. Either the solutions converged to a timing near the optimal predicted timing of approximately 1000μs, the period of the oscillator, near some integer multiple of the optimal timing, or they were in the vicinity of approaching such a timing as can be seen in the graph in Figure 4.1. This demonstrates that, with some further iterations included in the algorithm, the algorithm is capable of approaching optimal solutions for physical systems.

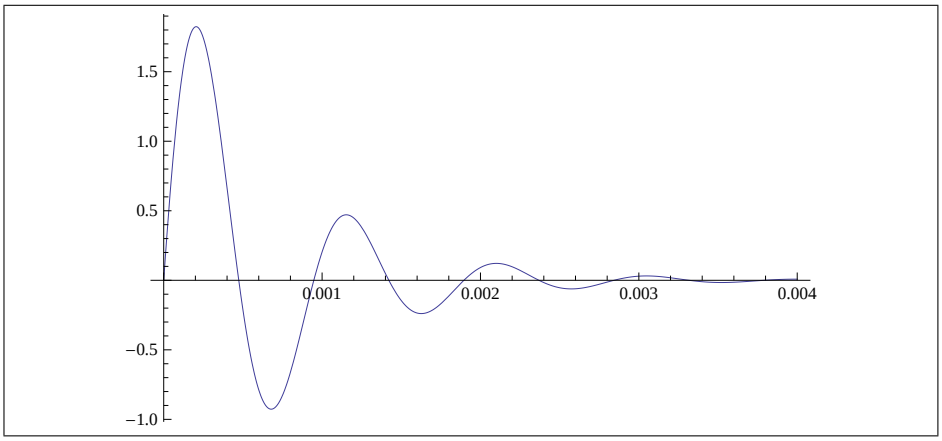

Figure 4.1: Plot of voltage for this LRC

However, this particular version of the experiment did not perform well at obtaining the necessary data for finding a fit for the potential energy. The charge data acquired from the timing was not good for determining a potential fit for this experiment. However, because we can obtain fitness amplitude measurements using this methodology there is still potential for this methodology to find the potential energy function if the timing can be assessed properly.

4.2 Next Steps With this Methodology

Some of the next steps with this particular methodology would be to test the various combinations of neighborhood selection and selection probability as was done in the simulations. Then it could be determined whether or not certain selections of this method are better or worse for this particular experiment.

In addition, this methodology either needs to be able to determine the timing at which a measurement was taken, or it needs to be applied to a different oscillator. This timing information is critical to determining the potential energy function, especially with regards to determining the charge. If it turns out that this particular experiment is not designed properly with regards to this particular oscillator, then it needs to be adjusted for a different oscillator. One example of a different oscillator that could be used is with a magnetic pendulum, and instead of finding the current voltage the experiment could keep track of the angular position.

4.3 Future Directions

Future applications of a similar methodology would involve various other types of oscillators that require different hardware and software capabilities, and for this methodology to be used in conjunction with another optimization method for when this method alone does not suffice for a particular problem.

One of the future applications intended for this method is to use it to find an optimal sequence of ultrafast laser pulses for exciting a particular crystal. These crystals have potential energy functions associated with them that are currently unknown. However, this experiment can be used to find the potential energy function using an analysis process similar to what is outlined in the motivation section.

However, for this particular problem sometimes it takes more than a few laser pulses before some sort of measurable oscillation happens. For this situation, it would be more than appropriate to use a genetic algorithm to find the optimal timings for the first few pulses. After that, this simulated annealing method can continue to build up the sequence pulse by pulse, allowing us to acquire the amplitude information after each pulse in the sequence afterwards. This will allow for us to determine the potential energy function of the crystal.

Appendix A

Simulated Annealing Test Algorithm

```
1  from pylab import * #For plotting
2  import math #For calling various functions
3  import random #For generation of pseudorandom numbers
4  import numpy as np #Numerical library
5  import scipy #Scientific computation library
6  import scipy.interpolate #For the interpolation function
7  import matplotlib.pyplot as plt #Also for plotting
8  import csv #For the generation of .csv files
9  import sys #For file writing
10
11 """
12 The following three lines define an interpolating function over
       the range
13 -10 to 10. The interpolation method is 1D cubic, and this function
        cannot
14 be evaluated outside of the bounds.
15 """
16 interpolationX = [-10.0, -9.0, -8.0, -7.0, -6.0, -5.0, -4.0, -3.0,
       -2.0, -1.0, 0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0,
       10.0]
17 interpolationY = [1.4125429788401118, 4.2303586635432469,
       2.9150883719169718, 2.4927172814567351, 1.0656216430970016,
       0.96046262410472871, 4.6972935801324649, 2.3895126263029107,
       2.3922138420600785, 2.3834152342501835, 1.4678720026242216,
       0.21532145496108546, 2.9089386977089138, 3.2503152700752054,
       0.87631644883654436, 2.3113950122881244, 4.8386630886391035,
       1.9046074692601027, 4.7934041954114956, 2.8450389093210138,
       1.3939939155628545]
18 interpolatedFunction = scipy.interpolate.interp1d(interpolationX,
       interpolationY,kind='cubic',bounds_error=True)
19
20
21 """
22 The following def commands establish a set of useful functions
23 for the simulated annealing algorithm.
24 """
25 #Defines the Boltzmann distrubution function for various
       probability computations
26 def boltzmann(x,y,T,k):
27     difference = y - x
```

```
28      divisor = k * T
29      p = float(difference) / float(divisor)
30      probability = math.exp(p)
31      return probability
32
33  #Defines the probability function from the standard Metropolis
        algorithm
34  def prob1(x,y,T,k):
35      if x < y:
36          return 1
37      else:
38          a = boltzmann(x,y,T,k)
39          return a
40
41  """
42  This defines an alternate probability function where better
        guesses are kept
43  most of the time, but not all of the time. This probability for
        the better
44  guesses is equal to one minus the probability of keeping a worse
        guess that
45  has the same difference in fitness from the current best guess.
46  """
47  def prob2(x,y,T,k):
48      if x < y:
49          a = boltzmann(y,x,T,k)
50          a = 1-a
51          return a
52      else:
53          a = boltzmann(x,y,T,k)
54          return a
55
56  #Allows the user to choose which probability function will be used
         during the round
57  def prob(x,y,T,k,choice):
58      if (choice == 'Metropolis'):
59          p = prob1(x,y,T,k)
60          return p
61      elif (choice == 'Alternate'):
62          p = prob2(x,y,T,k)
63          return p
64
65  #Gaussian fitness function for tests involving a single maximum
66  def f1(x):
67      p = math.pow(x,2)
68      p = -p
69      p = p / 25
70      fit = math.exp(p)
71      return fit
72
73  #The above interpolated function is called for tests involving
        multiple maxima
74  def f2(x):
75      fit = interpolatedFunction(x)
76      fit = float(fit)
77      return fit
78
79  #Lets the user select the fitness function for a given run
80  def f(x,choice):
81      if (choice == 'Gaussian'):
82          fit = f1(x)
83          return fit
```

```
84      elif (choice == 'Random Interpolation'):
85          fit = f2(x)
86          return fit
87
88  #Square root neighbourhood selection method
89  def limitFinder1(T,m):
90      a = T / m
91      a = math.sqrt(a)
92      return a
93
94  #Linear neighbourhood selection method
95  def limitFinder2(T,m):
96      a = m * T
97      return a
98
99  #Squared neighbourhood selection method
100 def limitFinder3(T,m):
101     a = T * T
102     a = a / m
103     return a
104
105 #Exponential neighbourhood selection method
106 def limitFinder4(T,m):
107     a = 1 / T
108     a = math.exp(-a)
109     a = m * a
110     return a
111
112 #Normal distribution neighbourhood selection method
113 def limitFinder5(T,m):
114     a = 2 * T * T
115     a = 1 / a
116     a = math.exp(-a)
117     tau = 2 * pi
118     s = math.sqrt(tau)
119     s = T * s
120     c = m / s
121     a = a * c
122     return a
123
124 #Allows the user to select which neighbourhood selection method to
        use
125 def limitFinder(T,m,intervalChoice):
126     if (intervalChoice == 'Constant'):
127         limit = m
128         return limit
129     elif (intervalChoice == 'Square Root'):
130         limit = limitFinder1(T,m)
131         return limit
132     elif (intervalChoice == 'Linear'):
133         limit = limitFinder2(T,m)
134         return limit
135     elif (intervalChoice == 'Squared'):
136         limit = limitFinder3(T,m)
137         return limit
138     elif (intervalChoice == 'Exponential'):
139         limit = limitFinder4(T,m)
140         return limit
141     elif (intervalChoice == 'Normal Distribution'):
142         limit = limitFinder5(T,m)
143         return limit
144
```

```
145 #Generates a new guess that is a "mutation" of the current best
        guess
146 def mutator(x,limit):
147     while (True):
148         while (True):#Does not allow a change of 0
149             change = random.uniform(-limit,limit)
150             if change != 0:
151                 break
152         new = float(x) + change
153         if (new < -10 or new > 10):#Only guesses within the bounds
                are chosen
154             continue
155         else:
156             break
157     return new
158
159 #Allows the user to choose the annealing schedule
160 def annealSchedule(T,const,choice):
161     if (choice == 'Exponential'):#Exponential annealing schedule
162         a = const * T
163         return a
164     elif (choice == 'Linear'):#Linear annealing schedule
165         a = T - const
166         return a
167
168 print "Set Temperature"
169 T = raw_input() #The user sets the temperature at the beginning of
        the run
170 print "Set Boltzmann Constant"
171 k = raw_input() #The user sets the Boltzmann Constant for the
        probability distribution function
172 print "Set Neighbourhood Constant"
173 m = raw_input() #The user sets the Neighbourhood Constant for a
        given problem
174 print "Choose Neighbourhood Selection Method (Constant, Square
        Root, Linear, Squared, Exponential or Normal Distribution)"
175 choice = raw_input() #The user sets the neighbourhood selection
        method
176 print "Set Schedule Constant"
177 scheduleConst = raw_input() #The user sets the constant for the
        annealing schedule
178 print "Choose Annealing Schedule (Exponential or Linear)"
179 scheduleChoice = raw_input() #The user chooses the annealing
        schedule
180 print "Choose Probability Method (Metropolis or Alternate)"
181 probChoice = raw_input() #The user chooses the selection
        probability method
182 print "Choose Fitness Function (Gaussian or Random Interpolation)"
183 fitChoice = raw_input() #The user selects which fitness function
        to explore
184 print "Set termination condition"
185 termin = raw_input() #The user sets the termination condition for
        the algorithm
186 print "Set file names"
187 fNameBase = raw_input() #The user sets the name base for the files
        generated
188
189 #Conversion to appropriate types from raw_input()
190 T = float(T)
191 k = float(k)
192 m = float(m)
193 choice = str(choice)
```

```
194 scheduleConst = float(scheduleConst)
195 scheduleChoice = str(scheduleChoice)
196 probChoice = str(probChoice)
197 fitChoice = str(fitChoice)
198 termin = float(termin)
199 fNameBase = str(fNameBase)
200 Tinitial = T #Saves the initial temperature for a log file
201
202 #Initializes the figure
203 fig = plt.figure()
204 ax = fig.add_subplot(211)
205 ay = fig.add_subplot(212)
206 x,y,z=[],[],[] #Empty lists for storing data
207 n=1 #Creates a counter
208
209 bestGuess = random.randint(-10,10) #Makes a random integer guess
        in the range
210 bestFitness = f(bestGuess,fitChoice) #Computes the fitness
211
212 #The following is the main simulated annealing loop
213 while (T > termin): #Runs for as long as T is greater than the
        termination condition
214     lim = limitFinder(T,m,choice) #Finds the limits of the
            neighbourhood
215     guess = mutator(bestGuess,lim) #Finds a new guess
216     fitness = f(guess,fitChoice) #Computes the new fitness
217     #Computes the probability of keeping the guess
218     maybeKeep = prob(bestFitness,fitness,T,k,probChoice)
219     test = random.random() #Generates a random float from 0 to 1
220     x.append(n) #Appends the current count to the list x
221     if (maybeKeep >= test): #If this test is true we keep the new
            guess
222         y.append(guess) #Saves the guess to the list y
223         z.append(fitness) #Saves the fitness to the list z
224         bestGuess = guess #The new guess is now the guess that is
                kept
225         bestFitness = fitness #The new fitness is saved
226     else: #If the test was false we disregard the new guess
227         y.append(bestGuess) #The current bestGuess is saved to y
228         z.append(bestFitness) #The current bestFitness is saved to
                z
229     n+=1 #Increments the count by 1
230     #Changes the temperature in accordance with the annealing
            schedule
231     T = annealSchedule(T,scheduleConst,scheduleChoice)
232
233 """
234 The following generates a .pdf with two graphs. The first graph
        represented by the
235 subplot ax is a graph of the guess with respect to the iteration.
        The second graph
236 represented by the subplot ay is a graph of the fitness with
        respect to the iteration.
237 """
238 ax.plot(x,y)
239 ax.grid()
240 ax.set_xlabel('iteration')
241 ax.set_ylabel('position')
242 ay.plot(x,z)
243 ay.grid()
244 ay.set_xlabel('iteration')
245 ay.set_ylabel('fitness')
```

```
246 ax.set_title('Simulated Annealing Test')
247 graphName = fNameBase + '.pdf'
248 plt.savefig(graphName)
249 #The following writes a .csv file with the data for further
        analysis in Mathematica if so desired
250 dataLogName = fNameBase + '.csv'
251 with open(dataLogName,'w') as dataFile:
252     #This estabilishes the formatting of the .csv and how entries
            are divided
253     dataWriter = csv.writer(dataFile, delimiter=',', quotechar='|'
            ,quoting=csv.QUOTE_MINIMAL)
254     c = 0
255     #This names the columns of the .csv
256     dataWriter.writerow(['Iteration','Best Guess','Fitness'])
257     while (c < len(y)):
258         #This grabs all elements of the lists and writes them to
                their
259         #respective columns in the .csv
260         guessString = str(y[c])
261         iterString = str(x[c])
262         fitnessString = str(z[c])
263         dataWriter.writerow([iterString,guessString,fitnessString
                ])
264         c+=1
265
266 #This shows the user the results of the run and allows the user to
        write
267 #commentary that will be saved in the logfile
268 print "The optimal input for our cost function is:"
269 print bestGuess
270 print "The optimal fitness is:"
271 print bestFitness
272 print "State Commentary Here"
273 Commentary = raw_input()
274 #All of this creates the string that will be written to the
        logfile
275 docString = 'The Temperature was: '
276 docString += str(Tinitial) + '\n'
277 docString += 'The Boltzmann Constant was: '
278 docString += str(k) + '\n'
279 docString += 'The Neighbourhood Constant was: '
280 docString += str(m) + '\n'
281 docString += 'The Neighbourhood Selection Method Chosen was: '
282 docString += choice + '\n'
283 docString += 'The Annealing Schedule was: '
284 docString += scheduleChoice + '\n'
285 docString += 'The relevant Annealing Constant was: '
286 docString += str(scheduleConst) + '\n'
287 docString += 'The probability method chosen was: '
288 docString += probChoice + '\n'
289 docString += 'The termination temperature set was: '
290 docString += str(termin) + '\n'
291 docString += 'The cost function chosen was: '
292 docString += fitChoice + '\n'
293 docString += 'The optimal input found was: '
294 docString += str(bestGuess) + '\n'
295 docString += 'The maximum value found was: '
296 docString += str(bestFitness) + '\n'
297 docString += 'Commentary:\n'
298 docString += str(Commentary)
299 docFileName = fNameBase + '.txt'
300
```

```
301 #This writes the logfile
302 with open(docFileName,'w') as logFile:
303     logFile.write(docString)
```

Appendix B

Main Experiment Python Code

```
#Imports the necessary libraries
import time
#For pauses
import serial
#For serial communication
import string
#For manipulating strings
from pylab import *
#For graphs
import math
#To include some key functions
import numpy as np
import scipy
import scipy.interpolate
import matplotlib.pyplot as plt
#For plotting
import csv
#For making .csv files
import sys
#For reading/writing files
import arduinoFitnessModule as AF
#For the fitness function
import random
#For generating random numbers

#Defines the boltzmann distribution function
def boltzmann(x,y,T):
    difference = y - x
    divisor = 0.245 * T
    p = float(difference) / divisor
    probability = math.exp(p)
    return probability

#Defines the Metropolis probability function
def prob1(x,y,T):
    if x < y:
        return 1
    else:
        a = boltzmann(x,y,T)
        return a
```

```
41
42 #Defines an alternate probability function
43 def prob2(x,y,T):
44     if x < y:
45         a = boltzmann(y,x,T)
46         a = 1-a
47         return a
48     else:
49         a = boltzmann(x,y,T)
50         return a
51
52 #Allows the user to choose between two probability functions
53 def prob(x,y,T,choice):
54     if (choice == 'Metropolis'):
55         p = prob1(x,y,T)
56         return p
57     elif (choice == 'Alternate'):
58         p = prob2(x,y,T)
59         return p
60
61 #Square Root selection method
62 def limitFinder1(T):
63     a = T / 0.000004
64     a = math.sqrt(a)
65     return a
66
67 #Linear selection method
68 def limitFinder2(T):
69     a = 150.0 * T
70     return a
71
72 #Squared selection method
73 def limitFinder3(T):
74     a = T * T
75     a = a / 0.067
76     return a
77
78 #Exponential selection method
79 def limitFinder4(T):
80     a = 1 / T
81     a = math.exp(-a)
82     a = 1657.76 * a
83     return a
84
85 #Defines the user's choice of neighborhood selection method
86 def limitFinder(T,intervalChoice):
87     if (intervalChoice == 'Constant'):
88         limit = 750.0
89         return limit
90     elif (intervalChoice == 'Square Root'):
91         limit = limitFinder1(T)
92         return limit
93     elif (intervalChoice == 'Linear'):
94         limit = limitFinder2(T)
95         return limit
96     elif (intervalChoice == 'Squared'):
97         limit = limitFinder3(T)
98         return limit
99     elif (intervalChoice == 'Exponential'):
100        limit = limitFinder4(T)
101        return limit
102
```

```
103 #Selects new solution from the neighborhood
104 def mutator(x,limit):
105     while (True):
106         while (True):
107             change = random.uniform(-limit,limit)
108             change = round(change, 0)
109             if change != 0:
110                 break
111         new = float(x) + change
112         new = int(new)
113         if (new < 100 or new > 6000):
114             continue
115         else:
116             break
117     return new
118 
119 #This is used in the writing of log files
120 def iteration(n):
121     if (n == 1):
122         return 'st'
123     elif (n == 2):
124         return 'nd'
125     elif (n == 3):
126         return 'rd'
127     else:
128         return 'th'
129 
130 #Defines the base temperature
131 Tbase = 10.0
132 #Lets the user select the neighborhood selection method
133 print "Choose Interval Selection Method (Constant, Square Root,
        Linear, Squared, or Exponential)"
134 choice = raw_input()
135 #Lets the user select the probability method
136 print "Choose Probability Method (Metropolis or Alternate)"
137 probChoice = raw_input()
138 #Lets the user set the file names for the experiment
139 print "Set file names"
140 fNameBase = raw_input()
141 #Defines how many timings in between pulses will be found
142 print "How many pulses in the pulse train?"
143 pulses = raw_input()
144 #Defines how many experiments the user wants to run
145 print "How many times do you want to repeat the experiment for?"
146 experiments = raw_input()
147 #Defines the number of points of data collected by the Arduino
148 print "How many points of data should the Arduino collect?"
149 dataPoints = raw_input()
150 #First part of the log file
151 docString = 'These were the selections made for this experiment:\n
        '
152 docString += 'The Neighbourhood Selection Method Chosen was: '
153 docString += choice + '\n'
154 docString += 'The probability method chosen was: '
155 docString += probChoice + '\n'
156 docString += 'The number of pulses per pulse train was: '
157 docString += pulses + '\n'
158 docString += 'The number of experiments ran was: '
159 docString += experiments + '\n'
160 docString += 'The number of points of data the Arduino collected
        was: '
161 docString += dataPoints + '\n'
```

```
162 #Converts some string inputs from the raw_input() function into
        the appropriate types
163 pulses = int(pulses)
164 experiments = int(experiments)
165 dataPoints = int(dataPoints)
166 #Lets the user know that the parameters are set
167 print "Parameters Set"
168 #Sets the termination condition
169 termin = 0.1
170 #This for loop runs for as many times as there are experiments
171 for i in xrange(0, experiments):
172     #Prepares initial guess
173     sol = []
174     experimentNumber = i + 1
175     #Lets the user know the experiment is beginning
176     print "Experiment beginning"
177     #This loop structure generates the pulse train
178     for j in xrange(0, pulses):
179         #The temperature is set to the base temperature
180         T = Tbase
181         #Selects a random guess from the solution space
182         bestGuess = mutator(3050,3000)
183         #Appends to the list sol the new guess
184         sol.append(bestGuess)
185         #Lets the user know the new guess is being tried
186         print "trying new guess"
187         #Finds the fitness
188         bestFitness = AF.arduinoFitness(sol, dataPoints)
189         n = 1
190         #initializes a plot
191         fig = plt.figure()
192         ax = fig.add_subplot(211)
193         ay = fig.add_subplot(212)
194         x,y,z=[],[],[]
195         #Lets the user know that a successful arduino
                communication has been made
196         print "new guess tried"
197         #This loop structure is the simulated annealing portion of
                the algorithm
198         while (T > termin):
199             #Finds the bounds of the neighborhood
200             lim = limitFinder(T,choice)
201             #Finds the new guess
202             guess = mutator(sol[j],lim)
203             #Creates a different list to try new solution in
204             solTrial = sol
205             solTrial[j] = guess
206             #Finds the fitness of the list
207             fitness = AF.arduinoFitness(solTrial,dataPoints)
208             #Establishes the selection probability
209             maybeKeep = prob(bestFitness,fitness,T,probChoice)
210             test = random.random()
211             x.append(n)
212             #If the test is successful, the new guess is the new
                    best solution
213             if (maybeKeep >= test):
214                 y.append(guess)
215                 z.append(fitness)
216                 bestGuess = guess
217                 bestFitness = fitness
218                 sol[j] = guess
219             #Otherwise, keep the old guess
```

```
220             else:
221                 y.append(bestGuess)
222                 z.append(bestFitness)
223             n+=1
224             #Decrease the temperature
225             T = T * 0.95
226         #Plots the graph of the selection process of the current
                list element
227         ax.plot(x,y)
228         ax.grid()
229         ax.set_xlabel('iteration')
230         ax.set_ylabel('position')
231         ay.plot(x,z)
232         ay.grid()
233         ay.set_xlabel('iteration')
234         ay.set_ylabel('fitness')
235         #Lets the user know the graph is prepared
236         print "graph prepared"
237         pulseNumber = j + 1
238         titleName = 'Slection process of element' + str(
                pulseNumber)
239         ax.set_title(titleName)
240         trialFileNames = fNameBase
241         trialFileNames += 'Experiment' + str(experimentNumber)
242         trialFileNames += 'PulseSpacing' + str(pulseNumber)
243         graphName = trialFileNames + '.pdf'
244         #Saves the graph
245         plt.savefig(graphName)
246         #Writes more of the logfile
247         docString += 'For the ' + str(i+1) + iteration(i+1)
248         docString += ' experiment, the optimal timing for the '
249         docString += str(j+1) + iteration(j+1)
250         docString += ' element of the sequence is: '
251         docString += str(bestGuess) + '\n'
252         docString += 'The corresponding maximum voltage attained
                for this is: '
253         docString += str(bestFitness) + '\n'
254         print "Writing Log Files"
255         dataLogName = trialFileNames + '.csv'
256         #Writes the raw data that was plotted in the graphs
257         with open(dataLogName,'w') as dataFile:
258             dataWriter = csv.writer(dataFile, delimiter=',',
                    quotechar='|',quoting=csv.QUOTE_MINIMAL)
259             c = 0
260             dataWriter.writerow(['Iteration','Best Guess','Fitness
                    '])
261             while (c < len(y)):
262                 guessString = str(y[c])
263                 iterString = str(x[c])
264                 fitnessString = str(z[c])
265                 dataWriter.writerow([iterString,guessString,
                        fitnessString])
266                 c+=1
267     #Writes the rest of the logfile for this experiment
268     docString += 'For the ' + str(i+1) + iteration(i+1)
269     docString += ' experiment, the optimal pulse train was: '
270     docString += str(sol) + '\n'
271     docString += 'The corresponding fitness for this pulse train
            was: '
272     docString += str(bestFitness) + '\n'
273     #Lets the user know that an experiment has been completed
274     print "Experiment done"
```

```
275 #Lets the user see the logfile and write commentary on it
276 print docString
277 print "State commentary here"
278 Commentary = raw_input()
279 docString += Commentary
280 docFileName = fNameBase + '.txt'
281 #Writes the logfile
282 with open(docFileName,'w') as logFile:
283     logFile.write(docString)
```

Appendix C

Arduino Module Code

```
#Import the serial library for interactions with the serial port
import serial
#Import the timing library for pauses
import time
#Import the string library to manipulate strings
import string

#Defines the fitness function
def arduinoFitness(List,points):
    #First we find the length of the list given to the function
    length = len(List)
    #Then we write the string with the length, timing list, and data points
    writeString = str(length) + str(List) + str(points)
    #Creates the serial object arduino for interaction with the Arduino
    arduino = serial.Serial(port='/dev/cu.usbmodem621', baudrate=115200, timeout=1)
    #Gives the Arduino time to turn on
    time.sleep(2)
    #Write the string to Arduino
    arduino.write(writeString)
    #Wait for experiment to complete
    time.sleep(0.5)
    d = 0
    dataList = []
    #This loop structure reads in the data
    while (d < points):
        readIn = arduino.readline()
        #The Arduino writes the characters '\r\n' at the end of each string it writes to the computer. This removes those characters.
        point = string.translate(readIn, None, deletions='\r\n')
        #Generates the list of data points
        point = int(point)
        dataList.append(point)
        d+=1
    #We want to maximize the minimum amplitude attained
    voltage = min(dataList)
    #This interprets the 10 bit number as a voltage
    voltage = float(voltage) / 1024.0
    voltage = voltage * 5
    voltage = voltage - 2.5
    voltage = abs(voltage)
```

```
40      #Close the serial object arduino
41      arduino.close()
42      #Return the voltage to the larger program
43      return voltage
```

Appendix D

Arduino Sketch Code

```
/* The setup() portion of the Arduino sketch defines
events that will happen upon the activation of the Arduino */
void setup()  {
  // Initialize the serial port with a baud rate of 115200
  Serial.begin(115200);
  // Set digital pin 5 to be an output
  pinMode(5, OUTPUT);
}

/* This function defines how a pulse is delivered
during the experiment. */
void pulse(int pin, int time)  {
  // Turns the pin on
  digitalWrite(pin, HIGH);
  // Delays for an amount of microseconds
  delayMicroseconds(time);
  // Turns the pin off after the time has passed
  digitalWrite(pin, LOW);
}

/* The loop() portion of the Arduino sketch defines
events that will happen repeatedly after the setup()
portion of the Arduino completes */
void loop()  {
  // Wait for something to arrive over the Serial port
  while (!Serial.available())  {
  }
  // Declare variables to store incoming data
  int length;
  int points;
  /* Read the first recognizable integer that
  comes in from the Serial port and save that
  to the variable length */
  length = Serial.parseInt();
  // Initialize an array to store times between pulses
  unsigned int array[length];
  // Declare an iterator i
  int i;
  /* This for loop writes to the array the times between
  pulses as generated by the Python program. The formatting
  of list variables in Python allows for the parseInt() command
  to be used to read in the times. */
  for (i = 0; i < length; i++)  {
    array[i] = Serial.parseInt();
```

```
45    }
46    /* Saves the last integer in the string as the number of data
47    points to collect */
48    points = Serial.parseInt();
49    //Creates an array to store the voltage data in
50    int data[points];
51    //Declares additional iterators
52    int d;
53    int c = 0;
54    /* This while loop first generates a pulse at each iteration
55    and then waits for a number of microseconds stored in the
56    cth element of the array. */
57    while (c < length)  {
58      pulse(5, 1);
59      delayMicroseconds(array[c]);
60      c++;
61    }
62    // Deliver one last pulse
63    pulse(5, 1);
64    /* This for loop reads in an amount of points equivalent
65    to the number defined earlier and saves them to the array
66    data[] */
67    for (d = 0; d < points; d++)  {
68      data[d] = analogRead(0);
69    }
70    //One final iterator
71    int e = 0;
72    /* This loop sends all the data points back up the serial port
73    as strings. These are then received and interpreted by the main
74    Python program */
75    while (e < points)  {
76      String dataString = "";
77      dataString += data[e];
78      Serial.println(dataString);
79      e++;
80    }
81  }
```

Appendix E

Mathematica Analysis for Experiment

```
L = 0.0035;
R = 10;
c = 6.5 10^-6;
qpulse = c 2.5 (1 - Exp[-((5 10^-6)/(R c))]);
charge[q_, t_] := q Exp[-((t R)/(2 L))] Cos[t/Sqrt[L c]]
voltage[n_] := Piecewise[{0.9814453125,n==1},{0.9716796875,n
    ==2},{0.83984375,n==3},{0.8642578125,n==4},{0.8837890625,n==5}]
timing[n_] := Piecewise[{1508 10^-6,n==1},{3983 10^-6,n==2},{2254
    10^-6,n==3},{3984 10^-6,n==4},{691 10^-6,n==5}]
data = {{qpulse + charge[qpulse, timing[1]], voltage[1]}};
AppendTo[data, {qpulse + charge[data[[2 - 1, 1]], timing[2]],
   voltage[2]}];
AppendTo[data, {qpulse + charge[data[[3 - 1, 1]], timing[3]],
   voltage[3]}];
AppendTo[data, {qpulse + charge[data[[4 - 1, 1]], timing[4]],
   voltage[4]}];
AppendTo[data, {qpulse + charge[data[[5 - 1, 1]], timing[5]],
   voltage[5]}];
sol = NonlinearModelFit[data, 1/2 k q^2, {k}, q];
Show[ListPlot[data], Plot[sol[q], {q, 0, 1.3 10^-6}]]
```

Bibliography

[1] Aarts, Emile and Korst, Jan *Simulated Annealing and Boltzmann Machines: A Stochastic Approach to Combinatorial Optimization and Neural Computing*, 1st Edition, Wiley, 1989.

[2] Fortney, Lloyd R. *Principles of Electronics: Analog and Digital*, Oxford University Press, USA, 2005.

[3] Griffiths, David J. *Introduction to Electrodynamics*, 3rd Edition, Addison Wesley, 1999.

[4] Hunter, John D. Matplotlib: A 2D Graphics Environment, *Computing in Science & Engineering* 9(3):90-95, 2007.

[5] Jones, Eric, Oliphant, Travis, Peterson, Pearu et. al. (2001), *SciPy: Open source scientific tools for Python*, [Online] `http://www.scipy.org/`

[6] Oliphant, Travis E. Python for Scientific Computing, *Computing in Science Engineering*. 9(3):10-20, 2007.

[7] *Python Programming Language-Official Website*, [Online]. `http://python.org/`, 2013.

[8] Symon, Keith R. *Mechanics*, 3rd Edition, Addison-Wesley, 1971.

[9] Thornton, Stephen T. and Marion, Jerry B., *Classical Dynamics of Particles and Systems*, 5th Edition, Brooks Cole, 2003.

[10] van Laarhoven, P.J. and Aarts, E.H. *Simulated Annealing: Theory and Applications*, Springer, 1987.

[11] Yao, Xin. *Dynamic Neighbourhood Size in Simulated Annealing*, volume 1 of *Proc. of Int'l Joint Conf. on Neural Networks (IJCNN'92)*, 411-416, 1992.