Some Notes from the Book: Practical Genetic Algorithms by Randy L. Haupt and Sue Ellen Haupt

John L. Weatherwax*

December 9, 2015

Introduction

Here you'll find some notes that I wrote up as I worked through this excellent book. I've worked hard to make these notes as good as I can, but I have no illusions that they are perfect. If you feel that that there is a better way to accomplish or explain an exercise or derivation presented in these notes; or that one or more of the explanations is unclear, incomplete, or misleading, please tell me. If you find an error of any kind – technical, grammatical, typographical, whatever – please tell me that, too. I'll gladly add to the acknowledgments in later printings the name of the first person to bring each problem to my attention.

Acknowledgments

Special thanks to (most recent comments are listed first): Christopher Ryan (found a bug in my tournament selection code), Arshia Prado (comments on the artificial neural network example that helped find a few bugs in the MATLAB codes) for helping to improve these notes and solutions.

All comments (no matter how small) are much appreciated. In fact, if you find these notes useful I would appreciate a contribution in the form of a solution to a problem that is not yet worked in these notes. Sort of a "take a penny, leave a penny" type of approach. Remember: pay it forward.

*wax@alum.mit.edu

Introduction to Optimization

Problem Solutions

Note: For many of these problems I used the Matlab function `testfunction.m` provided in the appendix of this book. Many of the local optimization routines discussed in this chapter assume an infinite domain for the independent variables of the minimization function. Since this is not true for all test function, if we are attempting to minimize a function defined over a bounded domain we need to either modify the local algorithms from their unbounded form to make sure that they respect variable constraints or *transform* the problem to a function defined on an unbounded domain using a technique like the logistic transformation as indicated in Problem 8 below. Once this transformation has been performed one can then use unbounded local methods to compute function minimizations.

Problem 1 (some classical optimization methods)

The various functions from the Appendix can be called using the Matlab function `testfunction.m`. If desired these functions can be plotted using the Matlab script `plot_test_functions.m`. Since various local optimization routines from this chapter require *derivative* information there are two Matlab functions denoted `testfunctionprime.m` and `testfunctionprimeprime.m` that calculate the first and second derivatives of the objective function respectively. Rather than implement the various local optimization methods discussed in the text we will simply use the Matlab optimization routines found here: `http://www4.ncsu.edu/~ctk/matlab_darts.html`. To use these routines we needed to use the same functional calling convention specified in the documentation with these tools. Thus we created a function "wrapper" around the above already mentioned test functions called `fn_wrap.m`. The optimization methods discussed in the text are implemented in the Matlab functions:

- `nelder.m` for Nelder Mead simplex optimization
- `bfgswopt.m` for the BFGS optimization method
- `steep.m` for the steepest decent optimization method

We choose to demonstrate the above solution methods on the `F10` function. This function is defined as

$$10N + \sum_{n=1}^{N} [x_n^2 - 10\cos(2\pi x_n)] \,, \tag{1}$$

which has an unbounded domain and has a unique minimum of 0 at the point $(x, y) = (0, 0)$ (when $N = 2$). For the function `F10` this exercise is worked in the Matlab script `chap_1_prob_1.m`. When that script is run we observe that depending on the starting point each algorithm can converge to points that are not the global minimum. The closer one starts

to the global minimum the better the algorithms do. If started close enough the algorithms generally converge to the global minimum.

Problem 2 (different starting values)

Different starting values can lead to very different optimal solutions as seen by the numerical experiments performed above.

Problem 3 (many different starting values)

We generate 25 random initial conditions and run the `steep.m` code for each one in the Matlab script `chap_1_prob_3.m`. The initial guess at the minimum is taken as `x0 = 10.*randn( 2, 1 );` or a Gaussian random vector centered at $(0, 0)$ with a standard deviation of 10. When we do that we get following

```
x_0=  11.65 y_0=    6.27 x_f=    12.93, y_f=     2.98, fn=  177.090
x_0=   0.75 y_0=    3.52 x_f=     3.98, y_f=    -1.99, fn=   19.899
x_0=  -6.97 y_0=   16.96 x_f=    -5.97, y_f=    -2.98, fn=   44.773
x_0=   0.59 y_0=   17.97 x_f=     2.98, y_f=    -1.99, fn=   12.934
x_0=   2.64 y_0=    8.72 x_f=    -2.98, y_f=    -2.98, fn=   17.909
x_0= -14.46 y_0=   -7.01 x_f=     1.99, y_f=     0.00, fn=    3.980
x_0=  12.46 y_0=   -6.39 x_f=     0.00, y_f=     9.95, fn=   99.492
x_0=   5.77 y_0=   -3.60 x_f=     5.97, y_f=     0.99, fn=   36.813
x_0=  -1.36 y_0=  -13.49 x_f=     4.97, y_f=     2.98, fn=   33.828
x_0= -12.70 y_0=    9.85 x_f=   -12.93, y_f=     9.95, fn=  267.627
x_0=  -0.45 y_0=   -7.99 x_f=     5.97, y_f=    -2.98, fn=   44.773
x_0=  -7.65 y_0=    8.62 x_f=     0.99, y_f=     0.99, fn=    1.990
x_0=  -0.56 y_0=    5.13 x_f=    -2.98, y_f=    -0.99, fn=    9.950
x_0=   3.97 y_0=    7.56 x_f=     3.98, y_f=     7.96, fn=   79.595
x_0=   4.00 y_0=  -13.41 x_f=     0.99, y_f=     5.97, fn=   36.813
x_0=   3.75 y_0=   11.25 x_f=     1.99, y_f=    -1.99, fn=    7.960
x_0=   7.29 y_0=  -23.77 x_f=     0.99, y_f=    -0.00, fn=    0.995
x_0=  -2.74 y_0=   -3.23 x_f=    -0.99, y_f=    -0.99, fn=    1.990
x_0=   3.18 y_0=   -5.11 x_f=    -0.99, y_f=    -3.98, fn=   16.914
x_0=  -0.02 y_0=   16.07 x_f=     2.98, y_f=    -3.98, fn=   24.874
x_0=   8.48 y_0=    2.68 x_f=     4.97, y_f=     4.97, fn=   49.747
x_0=  -9.23 y_0=   -0.70 x_f=     2.98, y_f=    -0.00, fn=    8.955
x_0=   1.48 y_0=   -5.57 x_f=    -0.99, y_f=    -0.99, fn=    1.990
x_0=  -3.37 y_0=    4.15 x_f=    -2.98, y_f=    -0.99, fn=    9.950
x_0=  15.58 y_0=  -24.44 x_f=    11.94, y_f=     7.96, fn=  206.941
```

We see that *none* of the solutions found the global minimum. If we decrease the standard deviation in the random variable draw to 1 from 10 we still don't discover the global min-

imum. If we further decrease it to 0.1 most optimization runs find the global minimum of $(0, 0)$. This is a good demonstration of how strongly dependent the initial condition of our minimization routines is.

Problem 5 (running with restarts)

This problem is implemented in the Matlab script `chap_1_prob_5.m`. For the minimization routines tested, this did not seem to be an effect that is worthy of consideration. Rerunning the minimization routines with an initial guess given by the solution to the previous iterate seemed to converge to the same optimum.

Problem 7 (using Lagrangian multipliers)

We first form the Lagrangian augmented function f_λ which in this case is given by

$$\begin{aligned} f_\lambda(u, v, w, x; \kappa_1, \kappa_2) &= u^2 + 2v^2 + w^2 + x^2 \\ &\quad + \kappa_1(u + 3v - w + x - 2) \\ &\quad + \kappa_2(2u - v + w + 2x - 4) \end{aligned}$$

We then compute the derivatives of this function with respect to all variables $(u, v, w, x, \kappa_1, \kappa_2)$ and find

$$\begin{aligned} \frac{\partial f_\lambda}{\partial u} &= 2u + \kappa_1 + 2\kappa_2 \\ \frac{\partial f_\lambda}{\partial v} &= 4v + 3\kappa_1 - \kappa_2 \\ \frac{\partial f_\lambda}{\partial w} &= 2w - \kappa_1 + \kappa_2 \\ \frac{\partial f_\lambda}{\partial x} &= 2x + \kappa_1 + 2\kappa_2 \\ \frac{\partial f_\lambda}{\partial \kappa_1} &= u + 3v - w + x - 2 \\ \frac{\partial f_\lambda}{\partial \kappa_2} &= 2u - v + w + 2x - 4\,. \end{aligned}$$

When we see these relationships equal to zero we get six equations in the six unknowns $(u, v, w, x, \kappa_1, \kappa_2)$. Since the above is a system of linear equations we can solve it easily. This is done in the Mathematica workbook `chap_1_prob_7.nb` where we get the solution

$$u = \frac{67}{69}\,, \quad v = \frac{2}{23}\,, \quad w = \frac{14}{69}\,, \quad x = \frac{67}{69}\,, \quad \kappa_1 = -\frac{26}{69}\,, \quad \kappa_2 = -\frac{18}{23}\,.$$

Problem 8 (transforms from a bounded domain to an infinite domain)

To perform the transformation suggested in the book we would use something like the *logistic function* defined by

$$P(t) = \frac{1}{1+e^{-t}}\,. \tag{2}$$

Then $p = P(t)$ maps the infinite interval in t i.e. $-\infty < t < +\infty$ to the bounded interval in p of $[0,1]$. To map the unbounded variable t to an arbitrary variable v in the interval $[a,b]$ we would use

$$v = (b-a)P(t) + a\,.$$

Thus to map the bounded domain $0 \le x \le 10$ and $0 \le y \le 10$ for the function `F7` to an infinite domain over new variables $\tilde{x}$ and $\tilde{y}$ we would use

$$x = 10P(\tilde{x}) \quad \text{and} \quad y = 10P(\tilde{y})\,,$$

to get the objective function

$$\tilde{f}(\tilde{x},\tilde{y}) = 10P(\tilde{x})\sin(40P(\tilde{x})) + 11P(\tilde{y})\sin(20P(\tilde{y}))\,.$$

This later objective function is more suited to optimization via the unbounded techniques above.

The Binary Genetic Algorithm

Notes on the Text

Notes on Variable Encoding and Decoding

I had trouble implementing exactly the expressions given for binary encoding of real variables. In particular I found that

- The `round` function should be the `ceiling` function. This can be deduced by recognizing what this procedure is "doing". Once we have a normalized variable q_{norm} such that $0 \leq q_{\text{norm}} \leq 1$ the first bit is denoting whether or not the value of q_{norm} is to the left (denoted by a zero) or to the right (denoted by a one) of the midpoint of the original interval $[0, 1]$ or $1/2$. Once we know which of the sub intervals $[0, 1/2]$ or $[1/2, 1]$ contains q_{norm} we then recursively split that intervals into two regions. The next bit determine whether the point q_{norm} falls to the left or right of the midpoint of this interval. The midpoint in this case will be $1/4$ if the point q_{norm} fall in $[0, 1/2]$ and will be $3/4$ if the point falls in $[1/2, 1]$. The procedure is recursively continued N_{gene} times.
- There should be no $2^{-(M+1)}$ in the definition of q_{quant} (at least it seems that encoding followed by decoding matches the original variables better without this term).
- The expression for q_n given by the books equation 2.9 or $gene \times Q^T$ is really the expression for the variable q_{quant}.

With these fixes I implemented binary encoding and decoding in the Matlab files: `var_encode.m` and `var_decode.m`. These functions uses the axillary function `enforce_bounds.m` and are tested to show that they work using the script `var_encode_decode_Script.m`. Running that script give the result shown in Figure 1.

An implementaion of a binary genetic algorithm

To work the various problems and to understand the material better I implemented a binary genetic algorithm in the MATLAB code `binary_GA.m`. An example of how to run this code is shown in the script `binary_GA_Script.m`. This code follows the discussion in the book quite closely. In the routine `select_mates.m` the user can choose from various types of ways to select the parents of the offspring:

- pairing from top to bottom
- random pairing

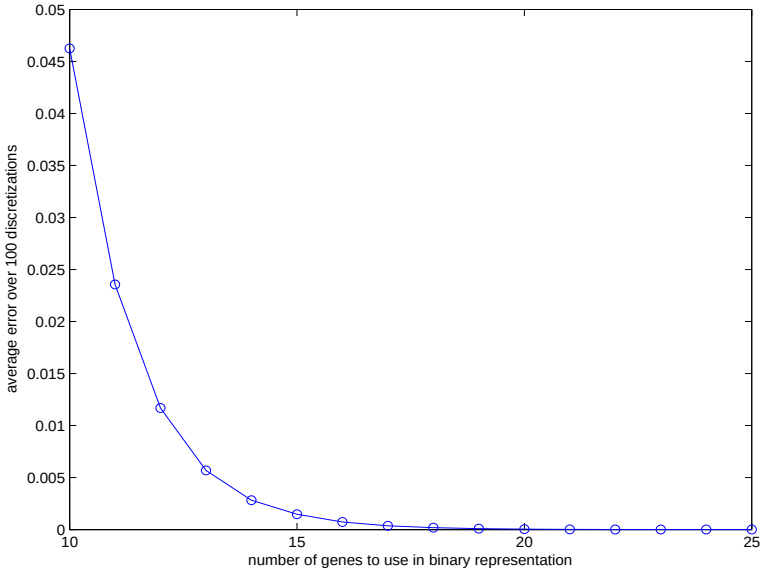

Figure 1: The results of running the script `var_encode_decode_Script.m`. For various values of N_{gene}, and for 100 trials, we generate 13 two dimensional random variables which we then encode using `var_encode.m` and decode using `var_decode.m`. For that trial we compute the matrix norm between the original samples and the encoded/decoded ones. The point plotted about N_{gene} is the average this norm over all of the 100 samples. We see that as we increase the number of genes the encoding/decoding procedure is more exact.

- rank weighting
- cost based weighting
- tournament selection

In the routine `crossover.m` the user can select various methods to perform crossover:

- single point
- double point
- uniform

The outputs from the routine `binary_GA.m` are the final binary population and their function values plus two arrays that given information on the algorithms convergence. The two arrays are the minimum and the average function values over the population at each timestep. We will use these routines to answer some of the questions below.

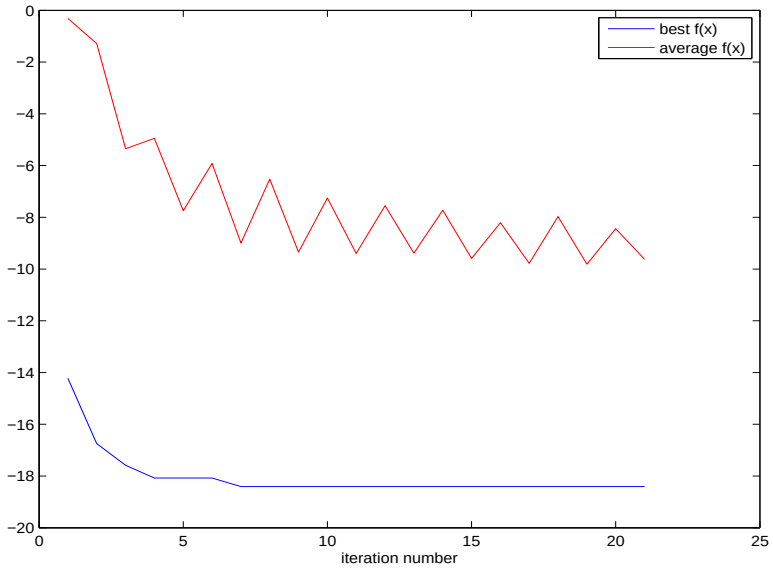

Figure 2: The results of running the script `binary_GA_Script.m` on the `F7` function.

Problem Solutions

Problem 1-2 (options for a binary GA)

These subroutines are all implemented in the discussed MATLAB files.

Problem 3 (using a binary GA to find the minimum of some functions)

In the routine `binary_GA_Script.m` we attempt to minimize a number of the functions given in the text book. For example if we attempt to minimize the function `F7` we get the following convergence plot given in Figure 2. We see that the smallest value of the objective function slowly making progress at getting smaller. The average fitness appears to oscillate as new genetic material is added from mutations. The value found for the minimum is given by

```
top 5 solution x values
    9.0759    8.7192
    9.0725    8.5866
    9.0725    8.5866
    9.0725    8.5866
    9.0725    8.5866

top 5 f(x) values
  -18.4062
  -18.3469
  -18.3469
```

```
-18.3469
-18.3469
```

These are very close to the optimum global value which can be found on Page 25.

Problem 4 (experiments with binary GA)

One thing that I found interesting when experimenting with these routines was that increasing the number of bits in a gene did not significantly "focus in" on the optimal location (or at least it did so very slowly). On the other hand, it seemed that with a genetic algorithm it was very "easy" to get "close" to the global minimization. What I mean by this is that depending on the problem sometimes with very small populations `N_pop=10` and very few bits `N_gene=10` the binary genetic algorithm was able to produce a solution very close the global optimum. This is effectively impossible with local optimization methods. These local optimization methods are able to converge to much greater precision however.

Problem 7 (sensitivity to μ and N_{pop})

Parts of this problem is worked in the MATLAB script `chap_2_prob_7.m` where we pick the function `F8` which seem to be rather difficult to find the exact minimum of. We then for various value of μ and N_{pop} we look at the smallest value of our objective function at the end of our algorithmic runs. We run our genetic algorithm 100 times saving the smallest value of our objective function each time. We then compute the average and the standard deviation of our minimum function value over all of these Monte Carlo trials. It is this average and its one sigma confidence interval that we then plot. These plots are presented in Figure 3. Further explanations are given in the figure caption. From the discussion in the caption it looks like having a larger population size is more important at ensuring we obtain the global minimum. The mutation rate needs to be large enough so that we continue to introduce genetic variability. Any larger and it just introduces variance to our optimization runs, meaning that each optimization run can give very different results. It might be wise then to start with a relatively small value for μ like 0.01 and only experiment with increasing its value after one has decided on a optimal value for N_{pop} (using plots like the above and looking for a "kink" in the graph).

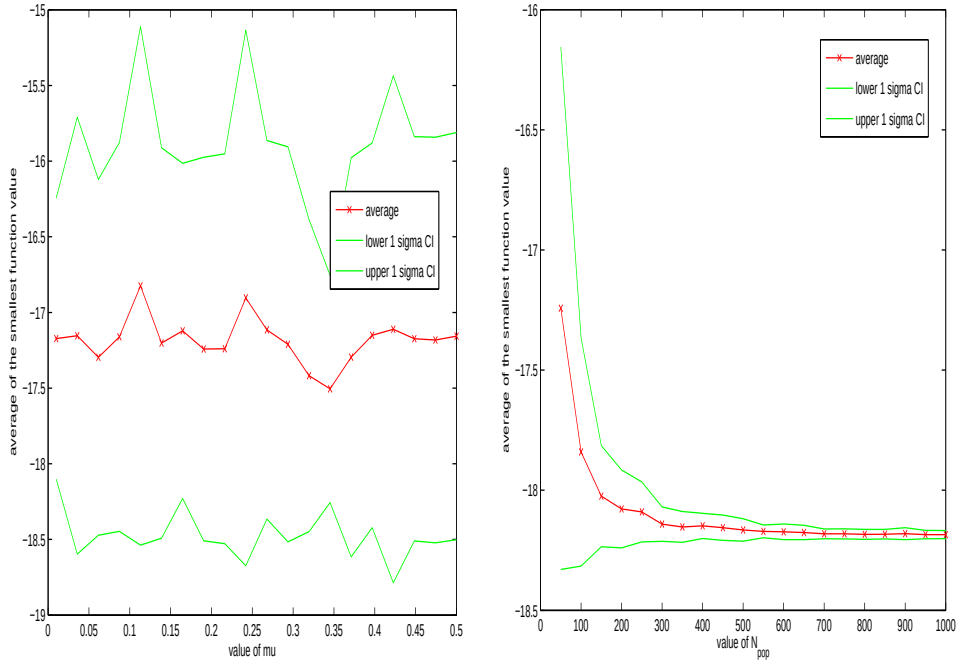

Figure 3: **Left:** The effect of the average minimum function (and its confidence interval) value found for the function `F8` using a genetic with $N_{\text{pop}} = 50$ for all runs as a function of the mutation rate μ. We see that as we increase the mutation rate the average of the smallest function value found can vary but is relatively flat. What is very large are the confidence bounds around this mean. Some of these optimization runs must find very low function values while others must not (to generate this large variance). **Right:** The effect of the average minimum function value found for the function `F8` using a genetic algorithm and confidence bounds on this value as a function of N_{pop}. We see that as we increase the size of the population the average of the smallest value found decreases quickly at first and then eventually flattens out. This plot indicates that for this problem if we run $N_{\text{pop}} > 600$ then we can be more sure that we will obtain the global minimum (since the variance is relatively small there).

The Continuous Genetic Algorithm

Notes on the Text

An implementation of a continuous genetic algorithm

To work the various problems and to understand the material better I implemented a continuous genetic algorithm in the MATLAB code `continuous_GA.m`. An example of how to run this code is shown in the script `continuous_GA_Script.m`. This code follows the discussion in the book quite closely. Just as in the binary genetic algorithm case in the routine `select_mates.m` the user can choose from various types of ways to select the parents of the offspring:

- pairing from top to bottom
- random pairing
- rank weighting
- cost based weighting
- tournament selection

In the routine `crossover.m` the user can select various methods to perform crossover:

- single point
- double point
- uniform
- various blending methods

The outputs from the routine `continuous_GA.m` are the final genetic population, their function values plus two arrays that given information on the algorithms convergence. The two arrays are the minimum and the average function values over the population at each timestep. We will use these routines to answer some of the questions below.

Problem Solutions

Problem 1-2 (options for a continuous GA)

Many of these subroutines are implemented in the discussed MATLAB files.

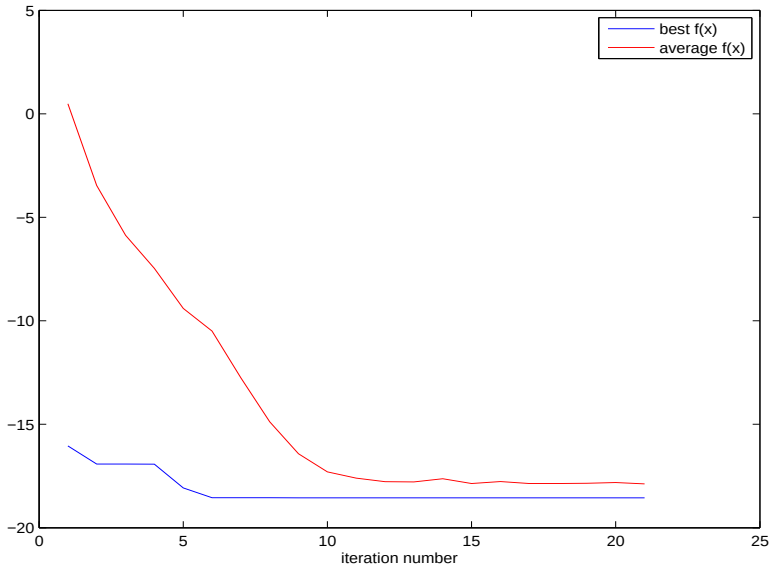

Figure 4: The results of running the script `continuous_GA_Script.m` on the `F7` function.

Problem 3 (continuous GA to find the minimum of some functions)

In the routine `continuous_GA_Script.m` we attempt to minimize a number of the functions given in the textbook. For example, if we attempt to minimize the function `F7` we get the convergence plot given in Figure 4. We see that the smallest value of the objective function slowly making progress at getting smaller. The average fitness appears to oscillate as new genetic material is added from mutations. The value found for the minimum is given by

```
op 5 solution x values
    9.0389    8.6682
    9.0389    8.6681
    9.0390    8.6681
    9.0389    8.6682
    9.0389    8.6681
top 5 f(x) values
  -18.5547
  -18.5547
  -18.5547
  -18.5547
  -18.5547
```

These are very close to the optimum global value which can be found on Page 25.

Problem 4 (experiments with continuous GAs)

This is similar to Problem 7 in the chapter on binary genetic algorithms. One could perform the same type of experiments here.

Problem 8 (differences between binary and continuous GAs)

One thing that I found interesting when experimenting with these routines was that the continuous genetic algorithm seamed more able to "focus in" on the optimal location as the number of iterations increased. It appeared that it was easier to obtain minimums with higher accuracy using the continuous genetic algorithm than using the binary genetic algorithm. Again, it was remarkable how "easy" it was to get "close" to the global minimization. What I mean by this is that depending on the problem sometimes with very small populations `N_pop=10` the continuous genetic algorithm was able to produce very quickly a solution very close the global optimum. This is effectively impossible with local optimization methods. To have a robust solution to a given problem one would need to study convergence plots like presented in Problem 7 in the previous chapter to determine values for N_{pop}, μ, the number of iterations, etc. that give consistent (and good) results.

An Added Level of Sophistication

Problem Solutions

Problem 1 (avoiding repeated chromosomes)

Part (a): As the initial population of a continuous genetic algorithm is much less likely to have duplicated elements we will only consider the case of a binary genetic algorithm. The book initially suggests the following Matlab code to generate the population of initial chromosomes

```
% Generate the initial population:
N_bits = N_gene*N_var;
pop    = round( rand(N_pop,N_bits) );
```

A drawback of this technique is that it can generate a population that has duplicate chromosomes. One suggestion given in the book to avoid this problem is to ensure that there is a unique binary string in each of the `N_pop` chromosomes. Note that if `N_pop` too large relative to `N_bits` it wont be possible to obtain unique chromosomes. In fact with `N_bits` we have at most $2^{\text{N\_bits}}$ unique binary strings. Thus we assume that $N_{\text{pop}} < 2^{N_{\text{bits}}}$. Given that constraint we can generate N_{pop} unique binary strings with the alternative commands

```
% Generate the initial population:
N_bits = N_gene*N_var;
max_N_BS = 2^N_bits;; % maximum number of binary strings
assert( N_pop < max_N_BS, 'unique chromosomes not possible' );
binary_strings = dec2bin( 0:(max_N_BS-1) );

% permute the order for more randomness
binary_strings = binary_strings( randperm(max_N_BS), : );

% split the char strings into arrays with numerical 0's and 1's
binary_numbers = zeros(size(binary_strings));
binary_numbers( binary_strings(:)=='1' ) = 1;

% a subset of these will be a unique binary population
pop = binary_numbers( 1:N_pop, : );
```

In the above code we generate "all" possible binary strings and then take a random selection as the initial population. If that takes too much computation time one could generate fewer elements of the `binary_strings` array.

Problem 2 (a method for dealing with multiobjective functions)

For this problem is based on the following observation that one could use to find the Pareto front (if we had infinite computer power). To begin with we will *specify* a fixed set of weights, w_n, such that $\sum_{n=1}^{N} w_n = 1$ from which we combine our N objective functions f_n as

$$cost(x) = \sum_{n=1}^{N} w_n f_n(x) \,. \tag{3}$$

We then run a standard genetic algorithm to find the optimal value for x^* that minimizes the above $cost(x)$ function. The point $(f_1(x^*), f_2(x^*), \cdots, f_N(x^*))$ is a point on the Pareto front. Note in a expression like this it is important that the range of each of the functions $f_n(x)$ be the *same*. If it is not then the optimization above may have trouble blending the functions $f_n(x)$ together. For the functions given here and where $1 \leq x_n \leq 2$ we should scale their ranges so that they all map to the limits $0 \leq f_n \leq 1$. This could be done in a number of ways. One way to do this would be for each of the above functions to tabulate its value some number of times for x_n between $[1, 2]$ and compute the minimum and maximum f_n value. We would then scale the outputs of f_n as

$$\hat{f}_n = \frac{f_n - f_{n,\min}}{f_{n,\max} - f_{n,\min}} \,. \tag{4}$$

We then implement the cost function as in Equation 3 but with $f_n \to \hat{f}_n$.

When there are many functions f_n, there are many possible values of w_n to compute minimization's over and this is a computational intensive method. When there are just *two* functions $f_n$the combination above becomes

$$cost(x) = w f_1(x) + (1 - w) f_2(x) \,, \tag{5}$$

with $0 \leq w \leq 1$, and we can perform a simpler discretization of w to compute the Pareto front. For this problem the book gives 9 weights for w for each of which we will run the genetic algorithm that we developed earlier to find the minimum of Equation 5.

We now compute the bounds for each of the functions f_1 and f_2 and then display our results. For the function f_1 given here defined by

$$f_1(x) = x_1 + x_2^2 + x_3 + \sqrt{x_4} \,,$$

by noting that each function in the sum is an increasing function of x_n we see that

$$\begin{aligned} f_1(x) &\leq 2 + 4 + 2 + \sqrt{2} = 8 + \sqrt{2} \\ f_1(x) &\geq 1 + 1 + 1 + \sqrt{1} = 4 \,. \end{aligned}$$

Thus we know the values of $f_{1,\min} = 4$ and $f_{1,\max} = 8 + \sqrt{2}$. For $f_2(x)$ defined by

$$f_2(x) = \frac{1}{x_1} + \frac{1}{x_2^2} + \sqrt{x_3} + \frac{1}{x_4} \,,$$

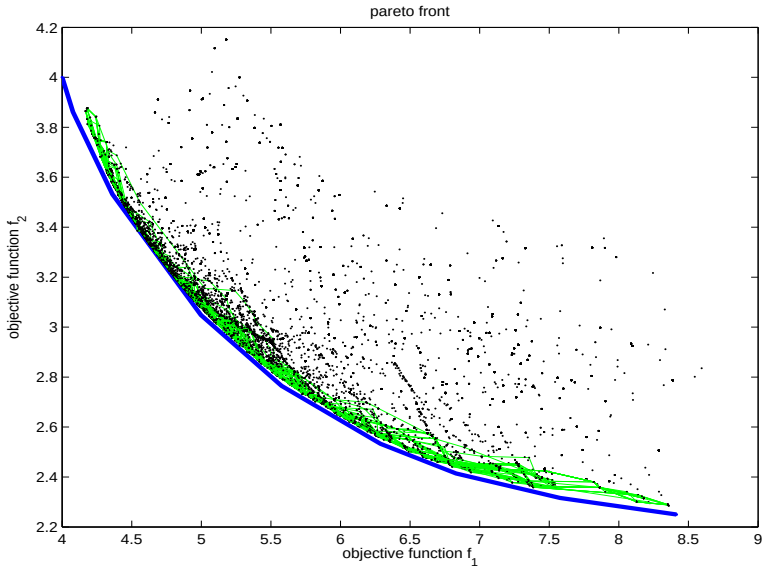

Figure 5: The results of running the script `chap_5_prob_2.m` followed by the script `pareto_GA_Script.m`.

since there are no cross product terms in its definition we see that

$$f_2(x) \leq 1 + 1\sqrt{2} + 1 = 3 + \sqrt{2}$$
$$f_2(x) \geq \frac{1}{2} + \frac{1}{4} + \sqrt{1} + \frac{1}{2} = \frac{9}{4}\,.$$

Thus we know the values of $f_{2,\min} = \frac{9}{4}$ and $f_{2,\max} = 3 + \sqrt{2}$. We first compute the Pareto front as discussed above using the MATLAB script `chap_5_prob_2.m` and associated routines. When that script is run the results are the blue line given in Figure 5.

Problem 3 (implementing a pareto genetic algorithm)

This is implemented in the MATLAB script `pareto_GA_Script.m` and the corresponding function `pareto_GA.m`. This later function is basically a coding of the function given in the appendix of the book. The "Script" function runs the pareto GA on the same objective function as the previous problem. When that script is run we get the result shown in Figure 5. That code will plot on each iteration the value of $(f_1(x), f_2(x))$ for each x in the population as a black dot. The pareto curve (for each iteration) is plotted as a green line. We see that as the Pareto GA converges the green lines become closer to the blue line computed in the previous problem. One wonders if we could use a technique like the previous problem to compute a few samples on the pareto front to help convergence.

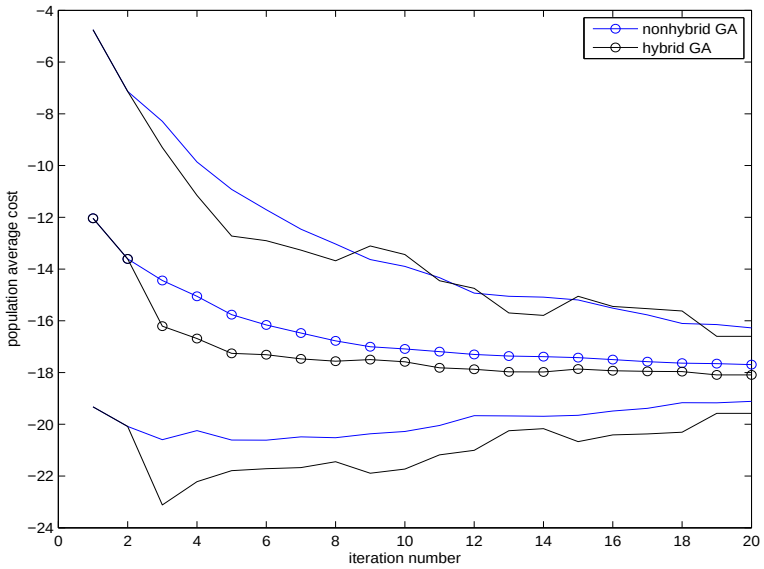

Figure 6: The results of running the script `hybrid_GA_Script.m`.

Problem 5 (a hybrid GA)

As discussed in the book, there are many ways one can implement a hybrid genetic algorithm. In this problem "hybrid" means we use genetic algorithms for a global minimization search and then use a local optimizer at some point in the optimization. I choose to implement a version of a hybrid genetic algorithm where every `hybrid_GA_n_iters` iterations (notionally 5) we use the Nelder-Mead simplex routine on some number `hybrid_GA_n_solutions` (notionally 3) solutions with the best fitness. The solution suggested by the genetic algorithm is only replaced/modified after the local optimization routine runs if the new solution has an objective function that is smaller than the one found by the GA. One needs to be careful when one codes these algorithms in that the local optimization routine can find optimum that are outside of the domain of interest and thus are not feasible solutions. As this change is really a slight change to the existing continuous GA developed earlier I will add this functionality as an option to that code. I then ran 100 monte carlo runs with initial random populations of size 16 and looked at the best solution at each iteration for both a standard GA and the hybrid GA in the Matlab script `hybrid_GA_Script.m`. When that script is run we get the result shown in Figure 6. This is for the 8th numerical optimization problem given in the text. In that figure one can see that the hybrid genetic algorithm is able to reach a lower cost much earlier in the sequence of iterates. This shows some of the power of applying this technique to your optimization routines.

Advanced Applications

Notes on the Text

Building Dynamic Inverse Models – The Linear Case

In this subsection of these notes we attempt to duplicate results from the book where they used a genetic algorithm to estimate the coefficients of in a linear dynamical system.

In the first method I choose to try to duplicate these results I tried to estimate the coefficients in the ordinary differential equation $\frac{d}{dt}\mathbf{x} = \mathbf{A}\mathbf{x} + \mathbf{b}$. Thus the coefficients we will try to estimate are the elements of the matrix $\mathbf{A}$ and the vector $\mathbf{b}$. To test this procedure we started with the *known* vector solution given by

$$x(t) = \begin{bmatrix} \sin(t) \\ \cos(t) \\ t \end{bmatrix} .$$

To have this be a solution the coefficient matrix $\mathbf{A}$ and the vector $\mathbf{b}$ for this must satisfy

$$\frac{dx(t)}{dt} = \begin{bmatrix} \cos(t) \\ -\sin(t) \\ 1 \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \begin{bmatrix} \sin(t) \\ \cos(t) \\ t \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} .$$

Thus the above vector for $x(t)$ will be a solution to the given dynamical system if we take

$$\begin{aligned} &a_{11} = a_{13} = 0\,, a_{12} = 1 \quad \text{and} \quad b_1 = 0 \\ &a_{22} = a_{23} = 0\,, a_{21} = -1 \quad \text{and} \quad b_2 = 0 \\ &a_{31} = a_{32} = a_{33} = 0 \quad \text{and} \quad b_3 = 1\,. \end{aligned}$$

Thus in this problem there are $12 = 9 + 3$ unknown parameters that the genetic algorithm must estimate. When I tried to estimate these parameters I found that after a large number of iterations using a continuous genetic algorithm that I couldn't get estimates of a_{ij} or b_i that looked anything like the truth. It was also true that the plots of the output variable $x_i(t)$ as a function of t for the approximate solutions looked nothing like the true a spiral that was the target output.

Because of this result I choose to try a second method. Since the solution for $z(t)$ is $z(t) = t$ we can write a differential equation for $z(t)$ directly in terms of a linear system of the form $\dot{x} = Ax$ by introducing the two components $z_1(t)$ and $z_2(t)$ defined as the value of $z(t)$ and its derivative. This is we introduce

$$\begin{aligned} z_1(t) &\equiv z(t) \\ z_2(t) &\equiv \dot{z}(t)\,. \end{aligned}$$

Then forming the vector $\mathbf{z} = \begin{bmatrix} z_1 \\ z_2 \end{bmatrix}$ we find that the differential equation for the vector $\mathbf{z}$ is given by

$$\frac{d}{dt}\mathbf{z} = \begin{bmatrix} \dot{z}(t) \\ \ddot{z}(t) \end{bmatrix} = \begin{bmatrix} z_2(t) \\ 0 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} z_1 \\ z_2 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \mathbf{z}\,.$$

Thus to incorporate the variables $x(t)$ and $y(t)$ we define the big vector $\mathbf{X}$ as

$$\mathbf{X} \equiv \begin{bmatrix} x(t) \\ y(t) \\ z(t) \\ \dot{z}(t) \end{bmatrix}\,.$$

Then from the above discussion the matrix differential equation we are looking to solve is given by

$$\frac{d}{dt}\mathbf{X} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} x(t) \\ y(t) \\ z(t) \\ \dot{z}(t) \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} \mathbf{X}\,.$$

To solve the above differential equation, we need an initial conditions on the vector $\mathbf{X}(t)$. To duplicate the books results We will take $x(0) = 0$, $y(0) = 1$, $z(0) = 0$ and $\dot{z}(0) = 1$. Thus the initial conditions we will use on the vector $\mathbf{X}$ are

$$\mathbf{X}(0) = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 1 \end{bmatrix}\,.$$

Note that for *linear* ordinary differential equations we can get the full solution using the fundamental matrix which is related to the matrix exponential e^{At}. A more general technique (that would work with nonlinear equations) is to solve the ordinary differential equation numerically.

To each of these problem formulations when we use the continuous genetic algorithm one needs to specify bounds on the variables that we are optimizing our objective function over. Since the known parameters are all ± 1 to make it more likely that the genetic algorithm will be able to find a solution I took these bounds at 2. It would be nice to be able to compute solutions with a larger number for this bound, implying that the genetic algorithm was in some sense robust to this choice.

In the three `Matlab` scripts `linear_inverse_model_Script.m`, `linear_ode_function.m`, and `genetic_ode_function.m`, I implemented these ideas using the genetic algorithm developed in previous chapters. One thing that was very prevalent was that solutions of the ordinary differential equation $\dot{x} = Ax$ can have exponentially growing behavior (not necessarily oscillatory). Because of that due to random mutation, we can get average population values that are quite large. The genetic algorithm seemed to stall, in that the best function evaluation didn't change much (or at all in subsequent iterations). It might be prudent to add more randomness if we don't see the best population solution improving. Based on the discussion

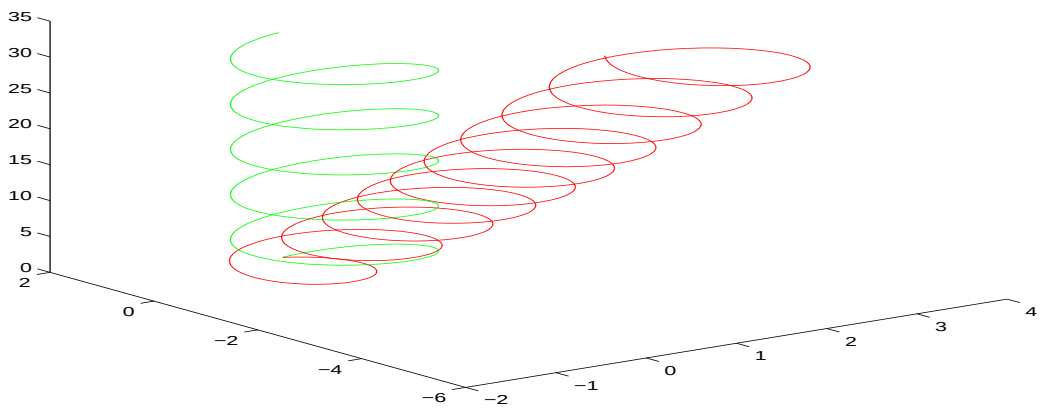

Figure 7: A first attempt at reproducing the linear inverse problem results from the book. The green curve is the true profile and the red curve is our approximate solution computed via a genetic algorithm. The qualitative behavior of the solution $\mathbf{X}$ has been captured.

above we obtained the following results when the Matlab code was run. First for the coefficient matrix computed via optimization A and the true coefficient matrix denoted by A_{truth} we get

```
A_truth =
     0     1     0     0
    -1     0     0     0
     0     0     0     1
     0     0     0     0
A =
   -0.1488    1.2380    0.1171   -0.1568
   -1.6300   -0.3665   -0.0909   -1.5075
    0.0031    1.0322    0.3169    1.4036
    0.8356    1.1452    0.0541    0.1281
```

These two results are not that great. There were done with a population size of 500 and 10000 iterations which is quite a lot of computations. One can see that its looks like the genetic algorithm is "almost" correctly estimating some of the coefficients. Which if we plot the output from these two solutions we get Figure 7.

In the previous attempt our optimization algorithm picked a matrix A, solved the ordinary differential equation (ODE) $\dot{\mathbf{X}} = A\mathbf{X}$, and then assigned a goodness of fit based on the difference between the ODE's solution and the known truth value for $\mathbf{X}$. From the above results this procedure seems to be slowly converging and perhaps with a great number of iterations or a larger population size would eventually converge. It seemed like it might be better to formulate the problem to use the fact that not only do we exactly know the values of $\mathbf{X}$ but we also exactly know the values of the *derivative* of $\mathbf{X}$. In fact if we assume that $\mathbf{X}$ has dynamics that are governed by an ordinary differential equation we have that $\dot{\mathbf{X}} = A\mathbf{X}$

for all t in the given range. That gives rise to a different criterion function that might be better at producing the correct value of A. What we do is to pick an A, and then using the *known* values for $\mathbf{X}$ and $\mathbf{X}_t$ compute the error that this value of A

$$||\mathbf{X}_t - A\mathbf{X}||^p \,,$$

where this norm has to be averaged over all of the sampling time points t_i. This is perhaps the exact norm that the book suggested in the first place and that I didn't understand how to use it. I implemented this function in the Matlab function `genetic_ode_function_xt_minus_Ax.m`. We obtained the following results when these code were run. The coefficient matrix computed via optimization and the true coefficient matrix denoted by A_{truth} we get

```
A_truth =
     0     1     0     0
    -1     0     0     0
     0     0     0     1
     0     0     0     0
A =
   -0.9729   -0.2285    0.1346   -1.9827
   -1.1279    0.0301   -0.0600    1.6432
    0.9579   -0.1003    0.0681   -0.6358
   -0.6355    0.8353   -0.0569    1.8645
```

Again these are not great results. The trajectory of $\mathbf{X}$ that this matrix generates does not look anything like that generated by the truth.

Warning: In short, I was *not* able to duplicate the results from the book for this section. If anyone can find something wrong with what I attempted to do (or agrees that they cannot duplicate them either) please contact me.

Notes on Optimizing Neural Nets with GA's

In this section I attempt to duplicate the results on using genetic algorithms to fit an artificial neural network. The function $f(x)$ considered in this section was

$$f(x) = \frac{12}{x^2\cos(x) + 1/x} \quad \text{for} \quad 1 \le x \le 5 \,,$$

which when plotted did not look anything like the plot given in the book. When I plotted the above function I get the graph in Figure 8 (left). A troubling problem is the the above function is singular at a point x in the domain $1 \le x \le 5$ which would certainly make the fitting process more difficult. Assuming that there was an error somewhere, I then tried to fit an artificial neural network with a genetic algorithms to a function which looked like the one suggested in the book. I choose the function

$$f(x) = \frac{10 - x^2\cos(x) - 1/x}{22} \,.$$

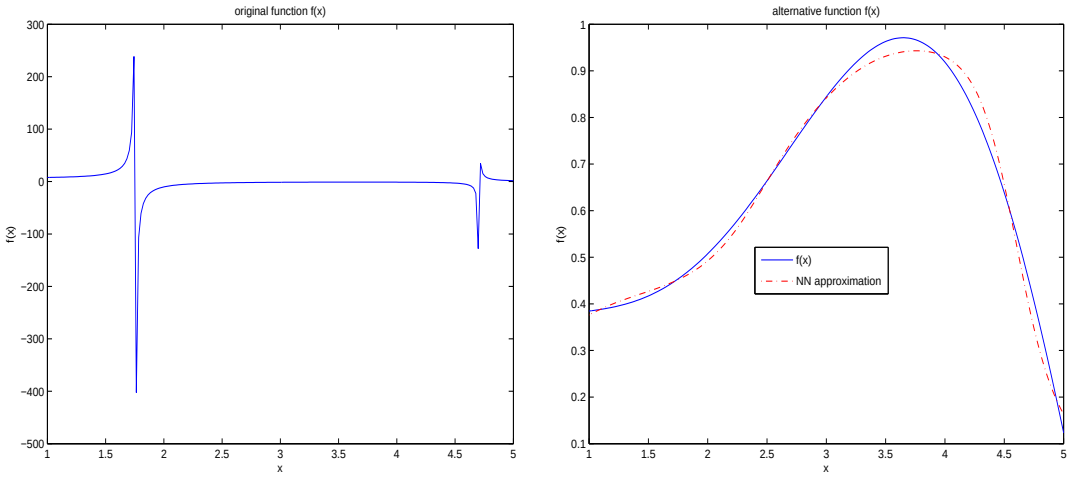

Figure 8: **Left:** The function $f(x)$ suggested in the book to be approximated with an artificial neural network (ANN). This function has singularity in the domain $1 \leq x \leq 5$. **Right:** The function $f(x)$ I attempted to fit with an ANN (blue) and the approximate function produced by the artificial neural network (in red).

This is plotted in the Figure 8 (right). Note that this function has its domain the interval $1 \leq x \leq 5$ and its range the interval $0 \leq f(x) \leq 1$. It is common in neural networks to requires all inputs and outputs to be bounded: either in the range $[0, 1]$ or $[-1, +1]$. Because of this I transformed the inputs using $\frac{x-3}{2}$ so that the natural domain $[1, 5]$ would be mapped to $[-1, +1]$. In the MATLAB script `artificial_neural_net_Script.m` we used a continuous genetic algorithm to find optimal weights and bias that should give the best function approximation to $f(x)$ when using a neural network. We can see the best function approximation found via the genetic algorithm in Figure 8 (right) as the red curve. In general, the agreement is quite good.

Notes on Solving High-Order Nonlinear Partial Differential Equations

For the ordinary differential equation

$$(\alpha u - c)u_X = \mu u_{XXX} - \nu u_{XXXXX} = 0\,,$$

If we let the solution $u(X)$ be parametrized as

$$u(X) = \sum_{k=1}^{K} a_k \cos(kX)\,,$$

we have the given derivatives

$$
\begin{aligned}
u_X &= \sum_{k=1}^{K} -a_k k \sin(kX) \\
u_{XX} &= \sum_{k=1}^{K} -a_k k^2 \cos(kX) \\
u_{XXX} &= \sum_{k=1}^{K} a_k k^3 \cos(kX) \\
u_{XXXX} &= \sum_{k=1}^{K} -a_k k^4 \cos(kX) \\
u_{XXXXX} &= \sum_{k=1}^{K} -a_k k^5 \sin(kX) \,.
\end{aligned}
$$

When we put that in the above we get

$$
\sum_{k=1}^{K} a_k \left(-k(\alpha u(X) - c) + \mu k^3 + \nu k^5\right) \sin(kX) = 0 \,.
$$

If we call the left-hand-side of the above expression as a cost, we see that this cost is X dependent via the $\sin(kX)$ *and* the function $u(X)$, given known values of a_k. Thus we can evaluate the left-hand-side of this expression at a discrete set of grid points and and then sum the absolute value of these points. If a_k is picked such that this cost is exactly zero we have found a solution to the original ordinary differential equation and to the super Korteweg–de Vries sKDV equation.

More Natural Optimization Algorithms

Notes on the Text

Notes on particle swarm optimization (PSO)

For particle swam optimization the velocity and a particle update are given by

$$v_{m,n}^{\text{new}} = v_{m,n}^{\text{old}} + \Gamma_1 r_1 (p_{m,n}^{\text{local}} - p_{m,n}^{\text{old}}) + \Gamma_2 r_2 (p_{m,n}^{\text{global best}} - p_{m,n}^{\text{old}}) \tag{6}$$

$$p_{m,n}^{\text{new}} = p_{m,n}^{\text{old}} + v_{m,n}^{\text{new}} . \tag{7}$$

Here r_1 and r_2 are random factors that change on iteration to iteration, the constant Γ_1 is the *cognitive* parameter that links the best local solution (found in the present population) to the current particle. The constant Γ_2 is the *social* parameter that links the best global solution to the current particle.

In the Matlab code `particle_swarm_optimization.m` we have implemented a very simple particle swarm optimization, that can allow the user to play with the parameters that are input into this algorithm.

Test Functions

the test functions for minimization

This book provides a nice explanation and description of genetic algorithms. It provides more than enough information for a practitioner to implement the ideas on his or her problem and get quite satisfactory results. It also provides a nice set of test functions at the end of the book that one can use to verify ones implementations. Unfortunately, it seems that there are a great number of typos in this section of the book. The obvious ones that I found were:

- The function `F3` defined as $\sum_{n=1}^{N} x_n^2$ is said to have a minimum at $(0,0)$ of 1 (it should be zero)
- The minimums of the `F7` and `F8` functions occur at exactly the same point and have exactly the same value (even thought the function definitions are different). If the suggested minimum point $(0.9039, 0.8668)$ is evaluated we get 0.5283 under `F7` and 0.5452 under `F8` neither of which is very close to the suggested minimum value of -18.55.
- The function `F5` defined as $\sum_{n=1}^{N} |x_n| - 10\cos(\sqrt{|10x_n|})$ is said to have a minimum at the scalar $x = 0$ with a value of 0.

There are probably other errors. To make sure that my genetic algorithm implemented when I worked through this book I implemented an exhaustive search for the minimum in the Matlab script `global_min_of_test_functions.m`. This could be improved in several ways perhaps by running a local optimization routine like discussed in this chapter on the point found from the exhaustive search I felt that having access to the true function minimization helpful to provide confidence that my genetic algorithm were working. We tabulate the minimum computed for some of the test functions here.

- `F1` Minimum at 0 of value 1.
- `F2` Minimum at 0 of value 0.
- `F3` Minimum at $(0,0)$ of value 0.
- `F4` Minimum at $(1,-1)$ of value 0.
- `F5` Minimum at $(0,0)$ of value -20.
- `F6` Minimum at (9.6200) of value -100.2237.
- `F7` Minimum at $(9.0389, 8.6679)$ of value -18.5547.
- `F8` Minimum at $(1.1781, 8.6939)$ of value -18.2004.
- `F9` Minimum varies depending on random draw.

- `F10` Minimum at $(0, 0)$ of value 0.
- `F11` Minimum at $(0, 0)$ of value 0.
- `F12` Minimum at $(-2.1417, -0.1515)$ of value -0.5231.
- `F13` Minimum at $(0, 0)$ of value 0.
- `F14` Minimum at $(1, 1.6606)$ of value -0.3356.
- `F15` Minimum at $(-2.7678, -5.0000)$ of value -16.9487.
- `F16` Minimum at $(-17.0077, 2.0742)$ of value -25.2305.

I have tried hard to make sure that everything above is correct but there could be some errors in these calculations. If anyone finds any please let me know.