Order from Chaos - A sampling of Stochastic Optimization Algorithms

A NOTE ON SOFTWARE USED IN THIS MODULE

This module is based on the OptLib library, which can be found on SourceForge at http://sourceforge.net/projects/optlib/. A manual can be found at http://sourceforge.net/projects/optlib/files/.

INTRODUCTION

Computational scientists often find themselves faced with the problem of finding the optimum solution to a problem with many possible parameters or input variables. Typical cases involve minimization, and as such optimization and minimization are often used synonymously, though finding a maximum would also be a form of optimization. Typical cases in the sciences might be finding the lowest energy configuration of a molecule, or finding the input parameters of a mode that result in model behavior that best matches data.

Optimization methods generally fall into one of two categories, deterministic or stochastic. Deterministic methods follow a simple set of rules, start with some initial guess, and then iteratively improve on the solution according to some algorithm—usually involving knowledge of the shape of the function at the current guess. Downhill simplex, method of steepest descent, and Powell's method (also referred to as the conjugate gradient method) are all deterministic methods of finding the minimum of a non-linear function of n variables. Deterministic methods for optimization have primarily one strength, and primarily one weakness. The strength of deterministic optimization methods is that given a good initial guess for an optimum solution they converge very rapidly and with high precision. The weakness is that they all, to a method, have a tendency to get trapped in local minima if the initial guess is far from the optimum solution.

Stochastic optimization methods, on the other hand, rely on randomness and re-trials to better sample the parameter space in searching for an optimum solution. The three most commonly used stochastic methods are Monte Carlo (MC), Simulated Annealing (SA), and the Genetic Algorithm (GA).

MC is pretty much what it sounds like, blind luck. In a Monte Carlo optimization, you simply guess randomly at a lot of things and see what you get. Variants of the MC algorithm include random walks, in which you choose a random new step at each iteration and test to see whether it is a better solution than the previous one, or hybrid MC/deterministic algorithms in which many random starting points are chosen as input to a deterministic method.

SA attempts to use a process analogous to the cooling of matter to find an optimum solution. As matter is heated, the increase in thermal energy (i.e., temperature) results in a large amount of vibration, and this allows the matter to exist in higher energy states, in a Boltzmann distribution. As matter cools, the vibration is removed and there is little change, and if it is cooled slowly, the matter will gradually return to its lowest energy state. In the SA algorithm, our vibration is modeled as a random walk with some step size Δx, and the function value is treated as the energy of the iteration. A random number

between 0 and 1 is generated, and if that number is less than $\exp(-\Delta E/T)$, the step is accepted. Many trials are made at a given temperature, and the step size (analogous to vibration) is increased or decreased such that any step is equally likely to be accepted or not. At this point, the optimization is said to be in "thermal equilibrium" and the temperature is then decreased. The two most important parameters controlling the progress of a SA minimization, then, are the initial temperature (which controls the degree to which a high energy step can be taken) and the cooling factor (which controls the amount of cooling applied when the minimization reaches thermal equilibrium for a given temperature). As the temperature is dropped, the step size will drop with it, and we can use the step size as a convergence criterion.

GA, on the other hand, uses a method analogous to evolution. Our input variables to the function being minimized are treated as if they were a strand of DNA—a genotype. Instead of a single guess, typically a variety of guesses are initiated and compared against each other at their ability to create a competitive phenotype (function value). The best genotypes are then recombined, by choosing random pairings and generating new genotypes that are based in some way on the pairings. This process is done iteratively until there is little variance in the phenotype of the population of genotypes.

Each of these methods has their strengths and weaknesses. Monte Carlo is very easy to code, but often requires a greater number of total function calls to optimize a problem with a high level of confidence. Genetic Algorithms have a poor ability to find a precise minimum, but are generally very efficient at getting close to the global minimum, and scale very well in parallel environments. GA has strong accuracy in a wide variety of situations but poor precision. Simulated Annealing methods have had varying success in their ability to scale on parallel environments. All of these are based on random numbers[1], and thus get slightly different answers each time, and must be run and analyzed as ensembles of optimizations, and the statistical meaning of occurrence rates within an ensemble is not in any clear way related to our usual understanding of predictive statistics, as it is not based on known samples or populations. All of this combines to give optimization a field with a reputation of being a "black art."

When applied and analyzed carefully, however, these approaches can be greatly successful at finding the minimum (or maximum) of a function.

Questions

[1] Random Number Generation itself is a discussion beyond the scope of this module, however there are many resources a student could use to learn more. http://www.shodor.org/cserd/Resources/Algorithms/RandomNumbers/ provides a discussion of the linear congruential generator, the primary method used in most RNGs as well as the problems that can occur. In short, RNGs do not actually create truly random numbers, but rather create a sequence of algorithmically generated "pseudo-random" values that given enough generations will repeat. This creates additional problems on a massively scaled system as the odds increase significantly that many processors using the same RNG algorithm might actually sample the same random numbers. Algorithms and implementations exist to help address this issue on massively scaled systems, such as the Scalable Parallel Random Number Generator (SPRNG, http://sprng.cs.fsu.edu/) code.

1. What is the origin of the phrase “Monte Carlo” modeling?
2. If optimization routines are typically set up to find the minimum of a function, what is the simplest way to modify a function in order to find the maximum instead?

AN EXAMPLE IN 2 VARIABLES

Consider the function (see Charbonneau 1995, Astrophysical Journal):

$$f(x,y) = \left[16x(1-x)y(1-y)\sin(n\pi x)\sin(n\pi y)\right]^2$$
$$x, y \in [0,1] \quad n = 1,2,...$$

which has the properties that it will have either 1 single global minimum at [0.5,0.5], or four degenerate global minimums near [0.5,0.5], with an increasing number of local minimums as n increases. (For n=1, there are no local minimums.)

For n=1, with a single well-defined minimum, a deterministic optimization routine such as Powell’s method will quickly find an optimum solution. As n increases, ensuring an optimum solution using a deterministic method becomes more difficult, even when coupled with a Monte Carlo approach.

For each of the following runs, an initial guess to Powell’s method of [0.75,0.75] was used.

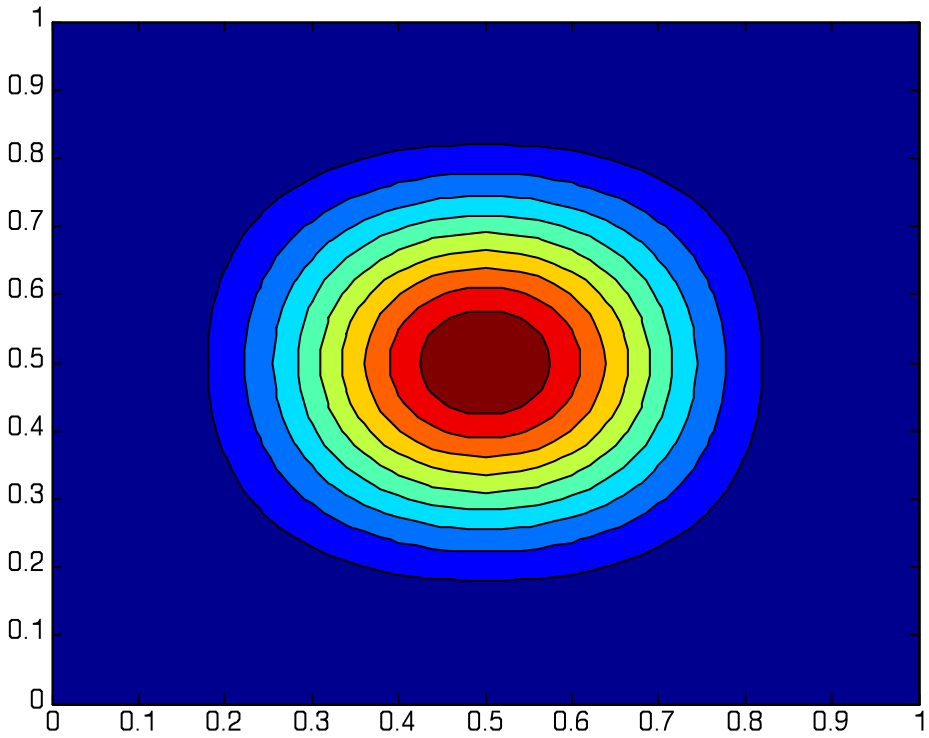
N	Solution Found	Function Calls Required
1	[5.000e-1,5.000e-1]	230
3	[7.945e-1,7.945e-1]	211
17	[7.345e-1,7.345e-1]	183

Note that the deterministic method in each case very quickly found the closest local minimum to the starting guess. This is typically what one expects with a deterministic method, and one would expect a similar result should we have used a method of steepest descent or a simplex method. Deterministic methods can find very accurate results quickly given a good starting guess.

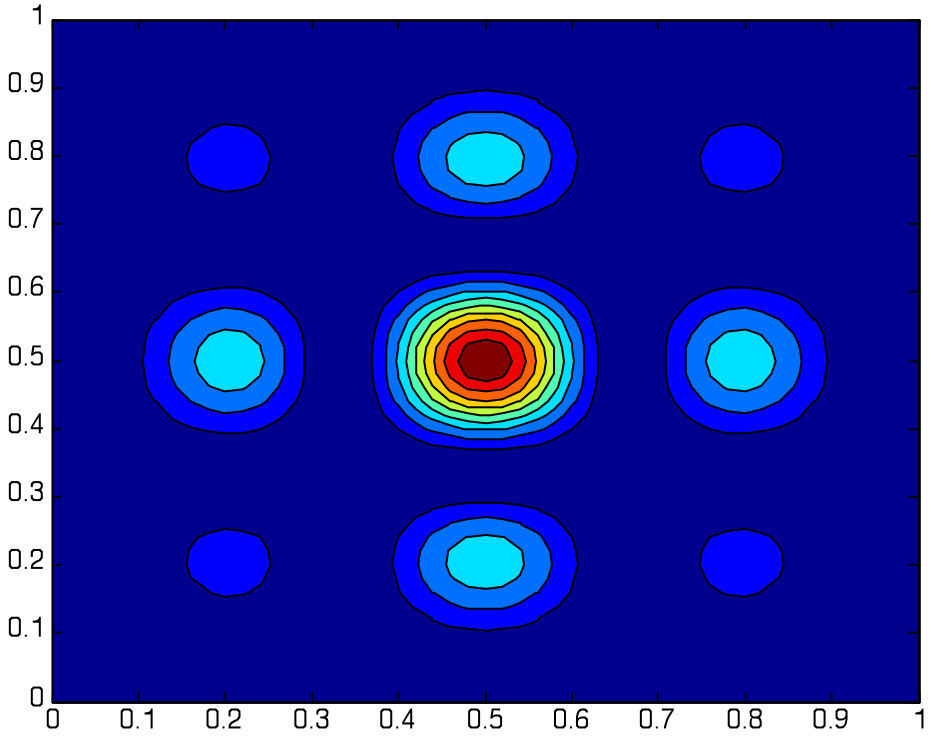
Stochastic methods will use pseudorandom numbers to attempt to better sample the space of possible local minimums.

Looking at the function being optimized for different n,

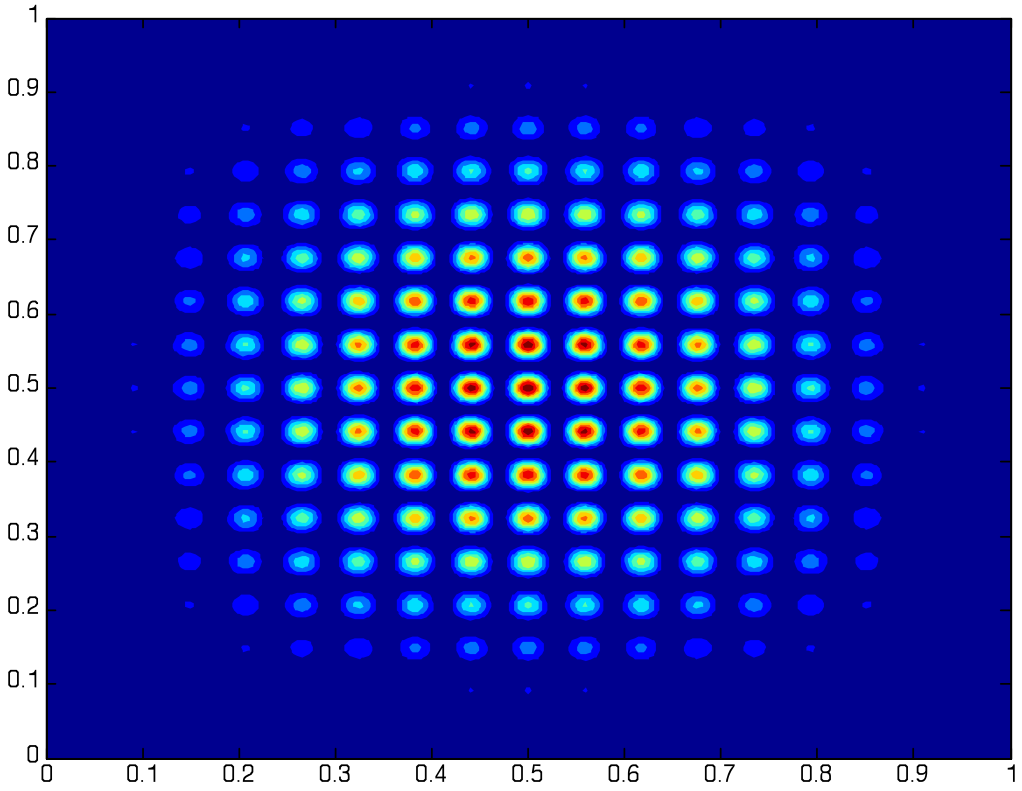
N=1



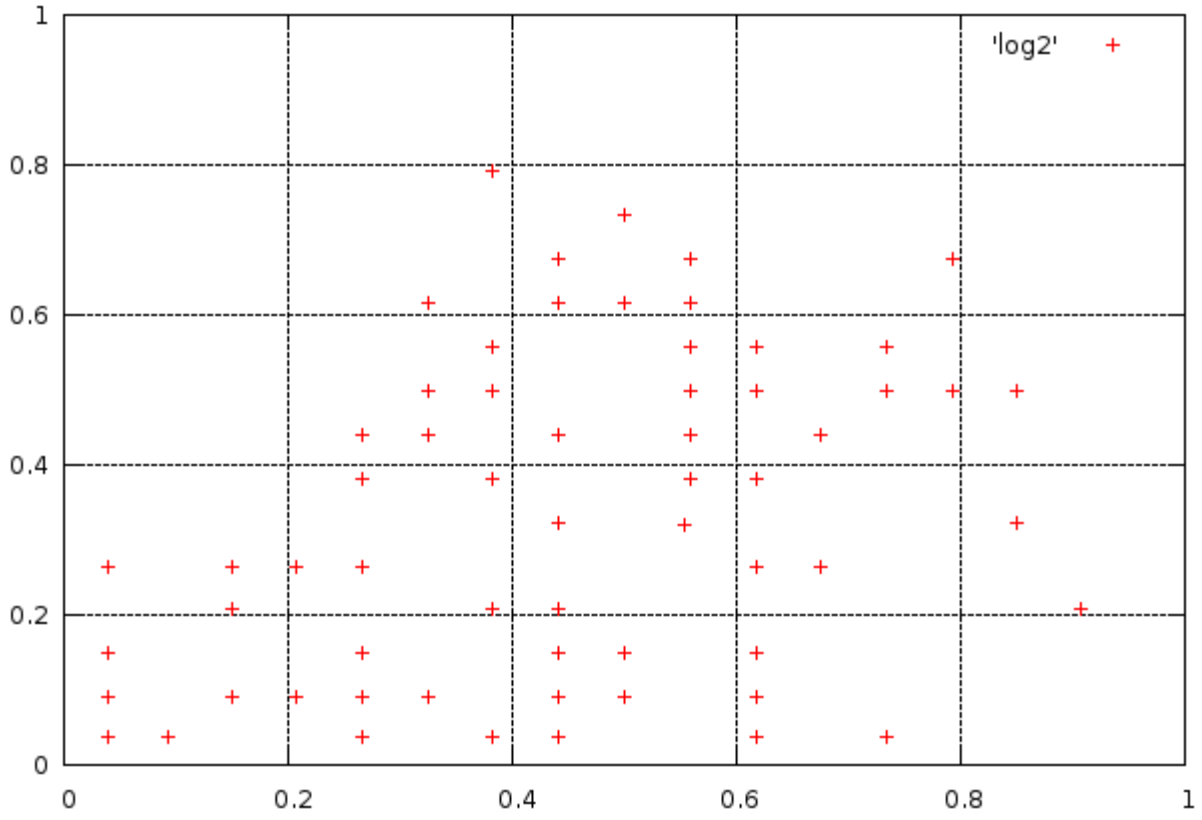
N=3



N=17



One possibility for finding the global minimum of this function would be to simply pick lots of random starting guesses, and feed that into a deterministic method. Suppose we picked 100 starting points at random and used each of those as input to a Powell's method optimization.



Note that in this case, the global optimum at 0.5,0.5 was not found in 100 random trials.

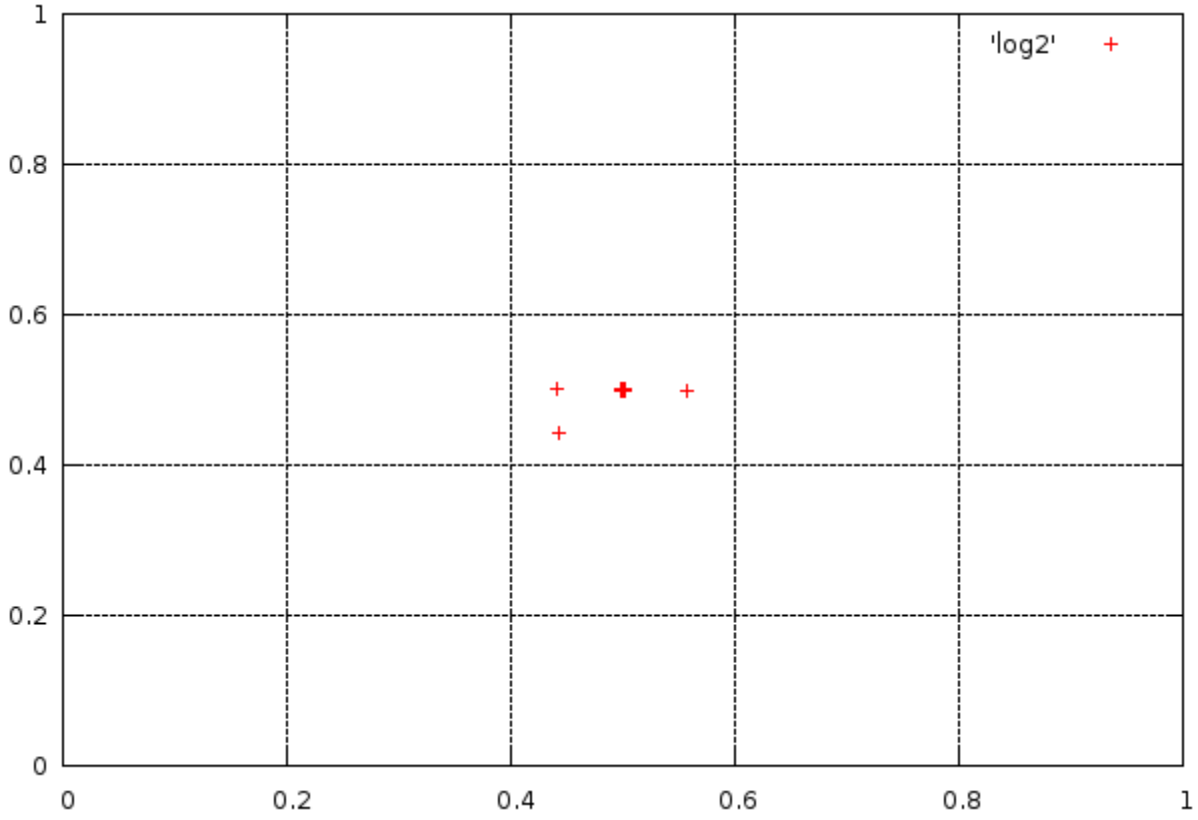
Another option might be to attempt a more directed stochastic approach. The two most commonly used today are simulated annealing, and genetic algorithms. In the case of simulated annealing, you pick a single guess at an optimum value, and then perform a random walk. The random walk is characterized by a "temperature" variable, a step size, and a "cooling rate." For each random step, steps in a direction that improve the optimization are always allowed, and steps that fail to improve the optimization are sometimes allowed, according to a Boltzmann distribution, where a random number is chosen between zero and one and compared to the value $\exp(-\Delta f / T)$, and if the random number is less than the exponential then the step is accepted. This is done for some set number of steps, and at each set number of steps the step size is updated such that about half of all steps are accepted. If too many steps are being accepted, the step size is increased, if too few, it is decreased. If about half are accepted, then the optimization is said to be in equilibrium and the temperature is decreased. The optimization converges when both the step size and temperature are "small."

Consider "typical" results using simulated annealing to solve the above problem—note that since random numbers are used one does not always get exactly the same result, so 3 trials will be shown for each value of N.

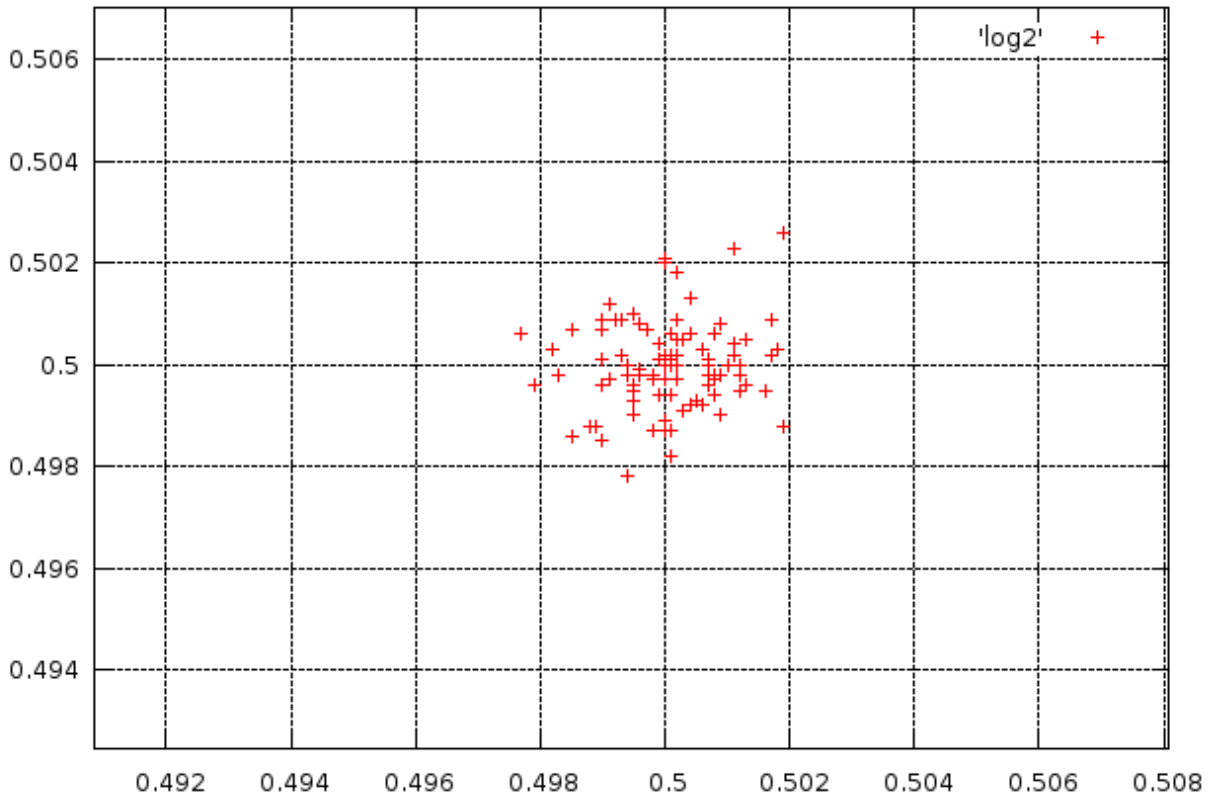
N	Solution Found	Function Calls Required
1	[5.000e-1,5.000e-1] [5.000e-1,5.000e-1] [5.000e-1,5.000e-1]	60097 59617 60385
3	[5.000e-1,5.000e-1] [5.000e-1,5.000e-1] [5.008e-1,5.000e-1]	59905 62113 121345
17	[4.995e-1,5.003e-1] [4.998e-1,4.999e-1] [4.988e-1,4.988e-1]	62017 60769 60865

Note that while simulated annealing did a much better job of accurately locating the global minimum, its precision, once found, was not as good as the Powell's method. Also, consider the number of function calls (individual evaluations of the function being minimized) required, which is typically 100s of times greater than for Powell's method.

We could look at the spread of 100 different attempts at using SA, compared to our Monte Carlo Powell's method with 100 different trials.



If one throws out the three "outliers," the above plot looks like



Another approach is to use a Genetic Algorithm. GA attempts to model the input parameters to an optimization as if they were a strand of genetic information, turning optimization attempt and value into genotype and phenotype, respectively. Many different “genotypes” are created at random and tested, and sorted in order of their ability to optimize the function. The best genotypes are used as input to create new genotypes to test, typically according to some mixing rule.

The implementation in OptLib follows the procedure listed below, and is performed in NP "tidal pools" separately. The best fits from each tidal pool are then used to create the initial population of a final minimization attempt:

- Determine initial random population g_i.
- For each g_i, solve for p_i, the weighted sum of square residuals between the observed spectra and a radiative transfer model spectra.
- Rank order g_i based on p_i, keep some fraction f_survival with lowest p_i for use in determining next generation.
- Recombine random pairings of surviving g_i using a combination of normal distributions.
- Calculate new values of p_i, repeat until p_i among surviving members of each successive generation converge to within some tolerance epsilon.

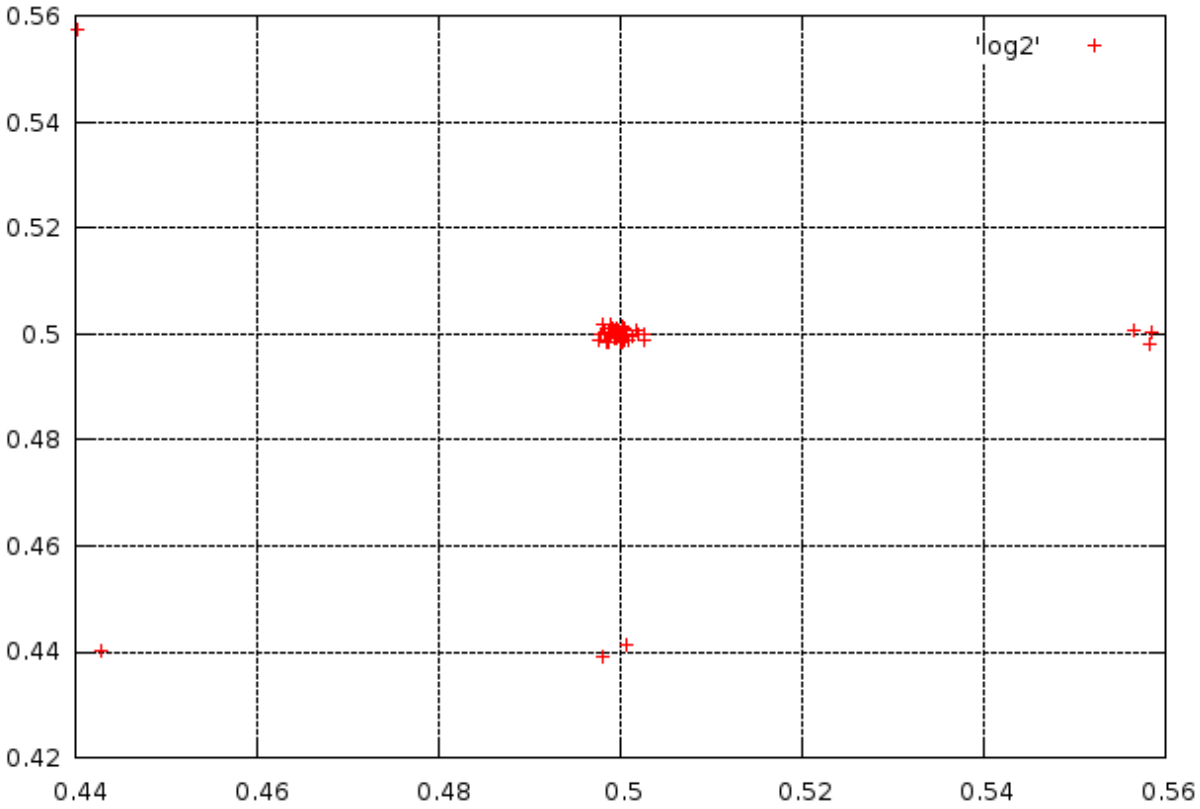
Recombination (or crossover) is determined using a probability distribution for each value of g_ij that allows for the possibility of traditional crossover where child values are close to one of the parent values (which we refer to as dominant recombination), the possibility of child values being close to the average of the two parents, and the possibility of child values being far from either parent value (which we refer to as mutation). This is modeled as the sum of 4 normal distributions, the relative weights of which are given by a dominace factor f_d/2 for each parent value of g_ij, a mutation factor f_m, and 1-(f_d+f_m) for averaged recombination. The standard deviations for each normal distribution are the minimum of the chosen parent value of g_ij or 1/10 of the range of the surviving g_ij values from the previous generation for dominant recombination, the maximum of the average parent value of g_ij or 10 times the range for mutation, or the range of surviving g_ij for averaged recombination. In our code, each of these standard deviations can be further modified by a step factor that is multiplied with the standard deviations-- this is generally used when change in the goodness of fit function and evolution for that parameter is desired to move slowly compared to other fit parameters. The stopping criterion for each tidal pool is defined as convergence in phenotype value to within some epsilon, typically with a minimum number of generations required to avoid early convergence to a local minimum. The mutation factor and the minimum number of generations can each be increased to avoid local minimums, the dominance factor can be combined with a low minimum number of generations to probe for local minimums--particularly in the case where a degenerate or near-degenerate solution is suspected.

The following were run using a population size of 500, with 50 allowed to mix each generation, and 10 pools that evolved separately before being mixed.

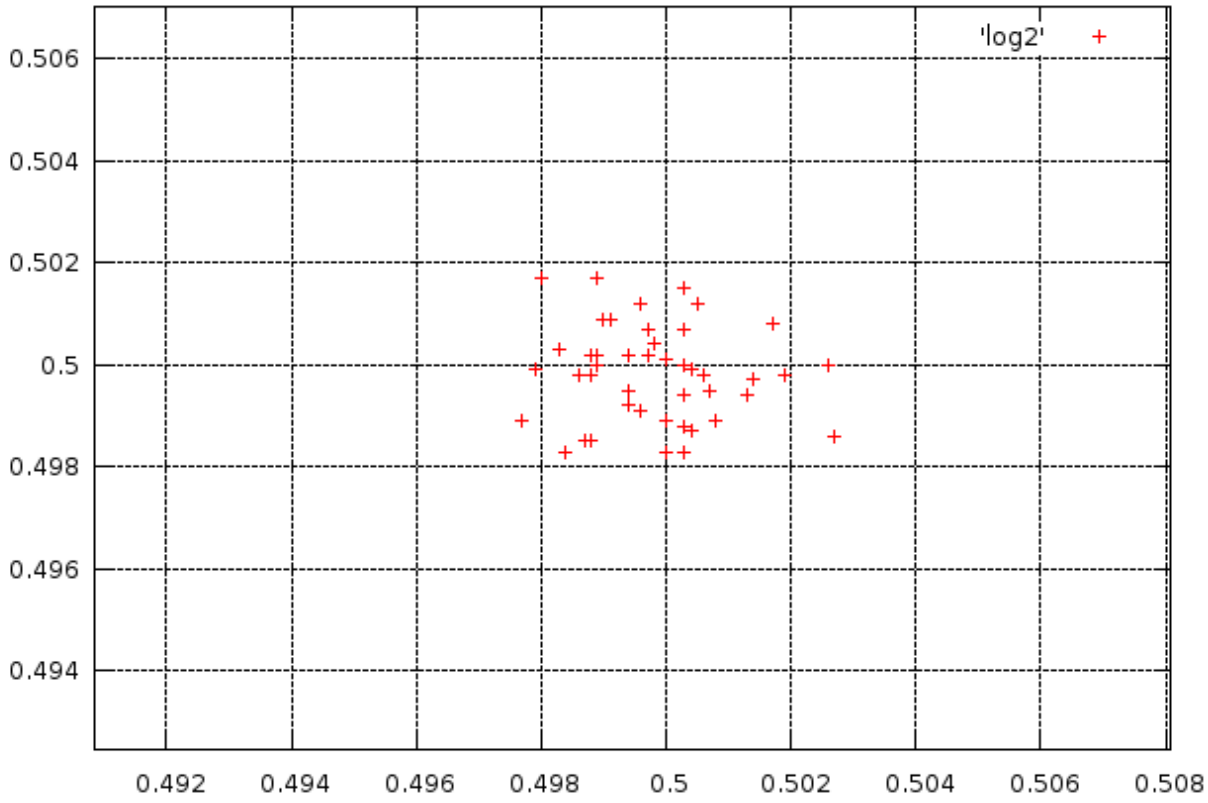
N	Solution Found	Function Calls Required
1	[5.006e-1,5.001e-1] [4.993e-1,4.996e-1] [4.980e-1,5.008e-1]	58000 66500 58500
3	[4.989e-1,5.007e-1] [4.982e-1,5.035e-1] [4.999e-1,4.996e-1]	62500 61000 63000
17	[5.003e-1,4.973e-1] [5.009e-1,4.995e-1] [4.414e-1,4.997e-1]	78000 61500 68000

Note that the genetic algorithms result has a harder time finding the exact minimum in the easy cases. It is often said that GA routines are very good at getting close to an optimal solution, but take a very long time to fully converge.

Again looking at 100 runs attempting to find the global minimum for N=17

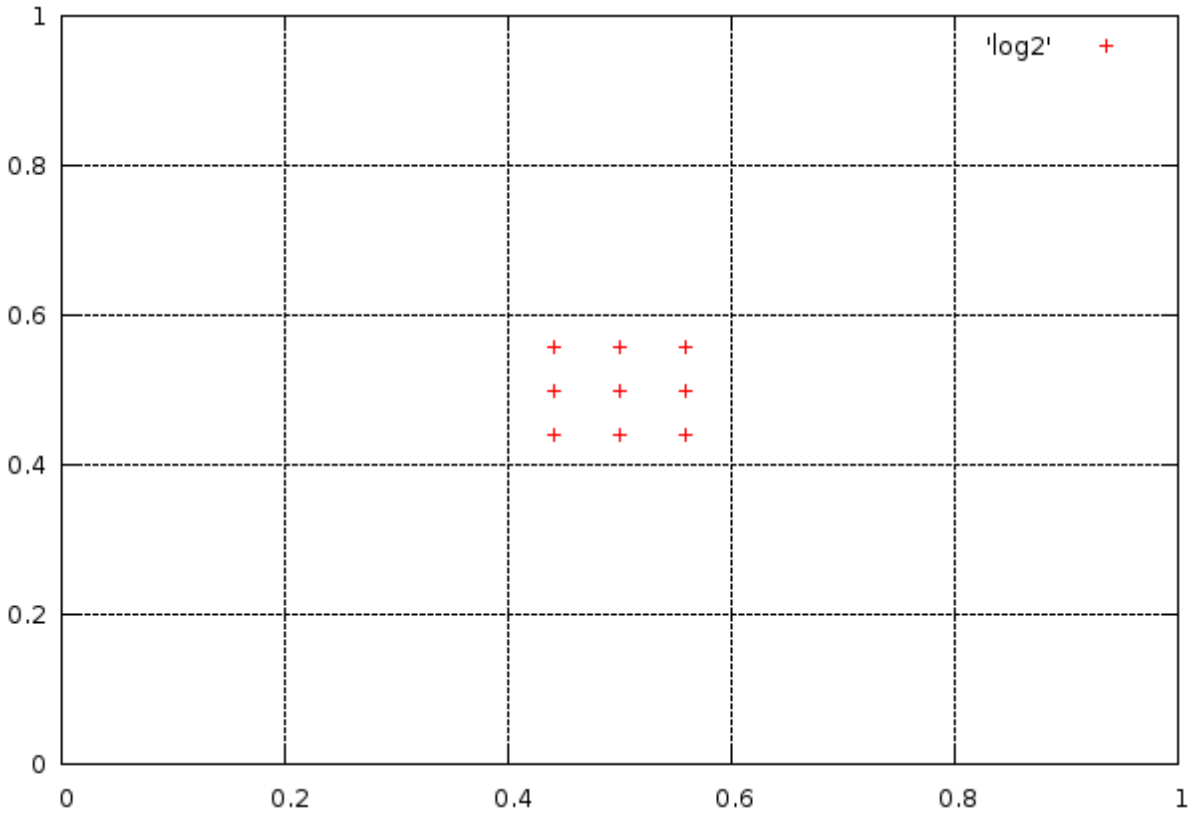


We see a slightly increased rate of outliers for this particular optimization, and looking at the central region, again once outliers are removed there is some spread about the optimum solution.



What we see in this case is that both the SA and GA methods are capable of picking out this particular global minimum with a failure rate of a few times out of a hundred, using roughly 60000-70000 function calls per optimization attempt.

Compare this to 100 attempts at using Monte Carlo-Powell for the same problem (N=17), with 100 trials per attempt (roughly the same number of function calls as our SA and GA approaches).



The Monte Carlo approach, can, if one has a reasonably good sense of where to look, find great detail in an optimization solution, whereas SA and GA can give you a reasonably good idea of where to look.

Exercises

1. Modify the function from Charbonneau to include any number of variables. Modify the func.c file included in the release of optlib 0.9 accordingly, and repeat the tests above for functions of higher numbers of variables.
2. How does the success rate (likelihood of an optimization attempt returning a correct global minimum) of each method change as the number of variables is increased?
3. How does the total number of function calls required per optimization change for each method as the number of variables is increased?
4. For a function with a single global minimum and multiple nearby local minimums of a similar value (such as the function above or a combination of sinusoidal functions of high frequency and a Gaussian function with a large width) compare each individual method to a coupling of GA or SA with MCP in which GA/SA gets close to the global minimum and MCP "cleans up" the result, as well as to an approach in which GA/SA is used to get close and a single call of a deterministic method such as Powell's is used to clean up the result. Discuss the trade-off between total number of function evaluations required and likelihood of finding the global minimum.

THE SIMSURFACE PROBLEM

Consider the case of 5 charged particles in a box with charged walls. Each particle is repelled by its neighbors but also by the walls of the box. One could calculate the potential due to Coulomb's law and minimize the potential energy of the system to determine the most likely configuration of the system:

$$V = \sum_{i=1}^{n} q_i \left[\sum_{j=i+1}^{n} \frac{q_j}{r_{ij}} + \sum_{k=1}^{4} Q_k \ln\left(\frac{\sqrt{d_k^{\,2} + b_k^{\,2}} + b_k}{\sqrt{d_k^{\,2} + a_k^{\,2}} - a_k} \right) \right]$$

where q_j is the charge of the j th particle and Q_k is the charge of the k th wall. d_k is the perpendicular distance to the k th wall, and a_k and b_k are the parallel distances to the k th wall for the i th particle.

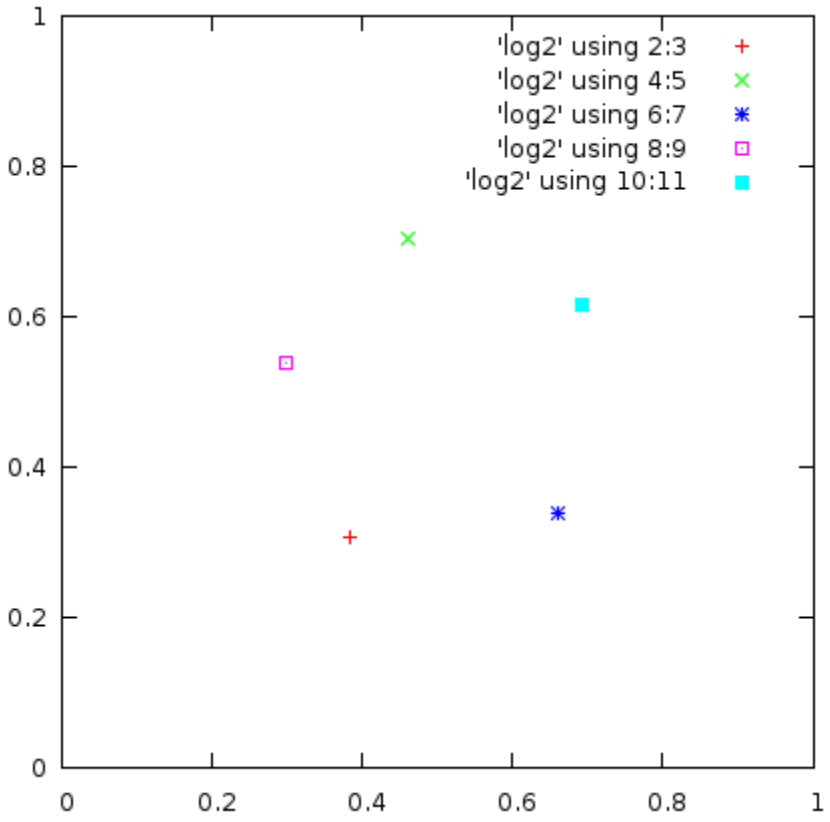
(Note that this is a unit system where the charges are in a box extending from 0,0 to 1,1, and all constants are equal to 1.)

You can look at simple systems of this sort using the tool at www.shodor.org,

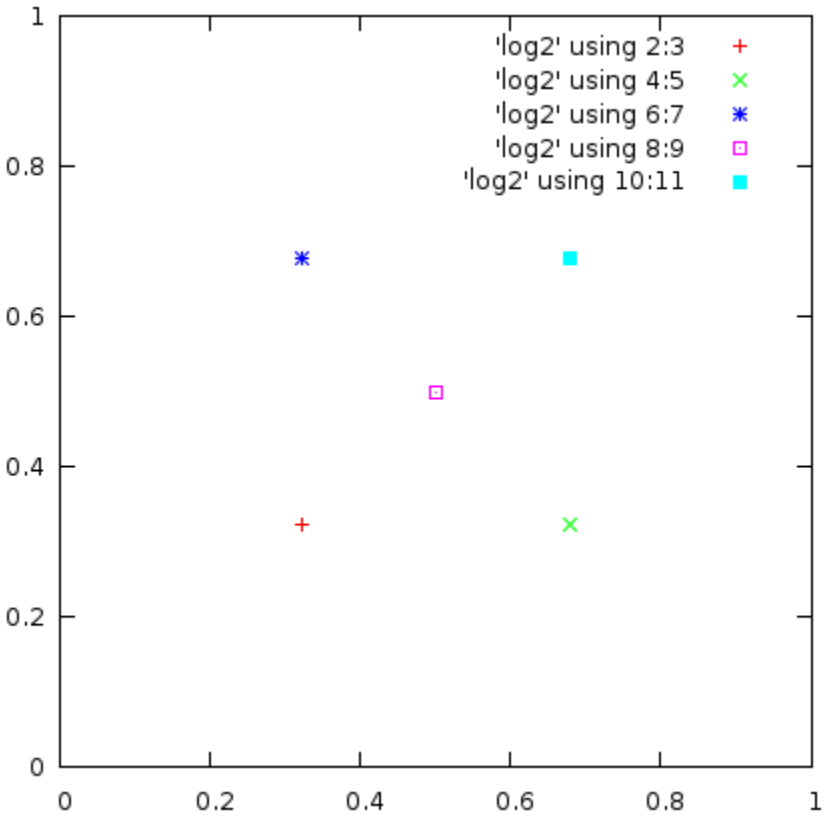
http://www.shodor.org/master/simsurface/

This tool will perform a simulated annealing (single annealer, not ensemble based) solution of n charges in a box of the nature described above.

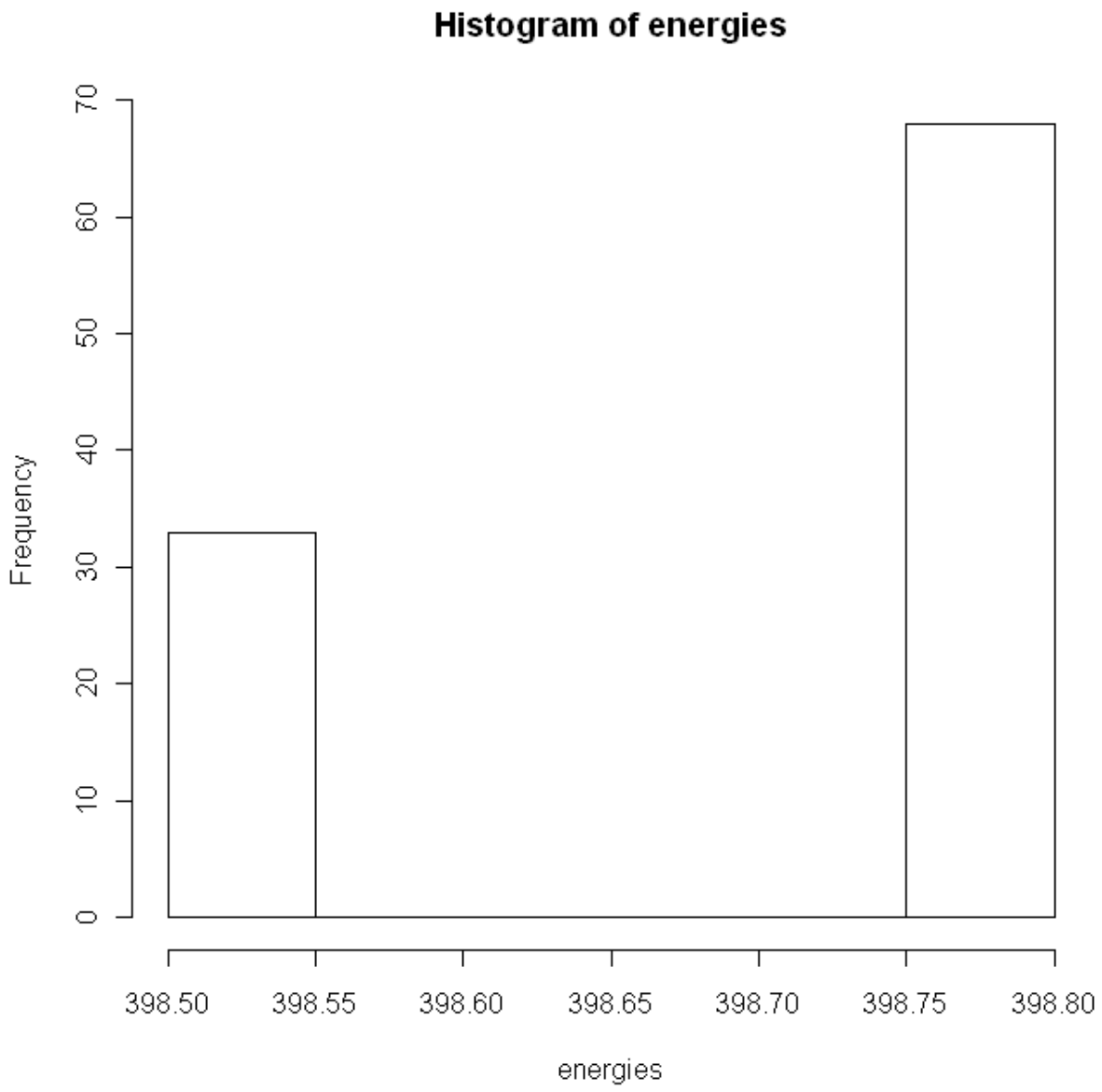
Typical results for 5 charges will depend on the strength of the walls, but will show two minimums with very similar potential energies as follows


'log2' using 2:3
'log2' using 4:5
'log2' using 6:7
'log2' using 8:9
'log2' using 10:11
1
0.8
0.6
0.4
0.2
0
0
0.2
0.4
0.6
0.8
1

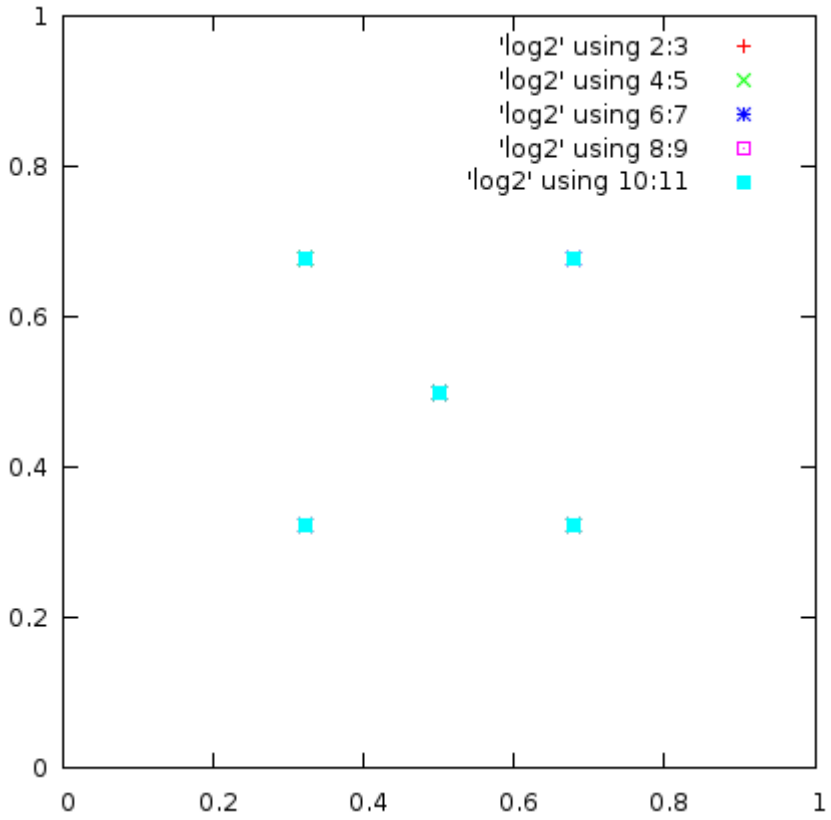
Or



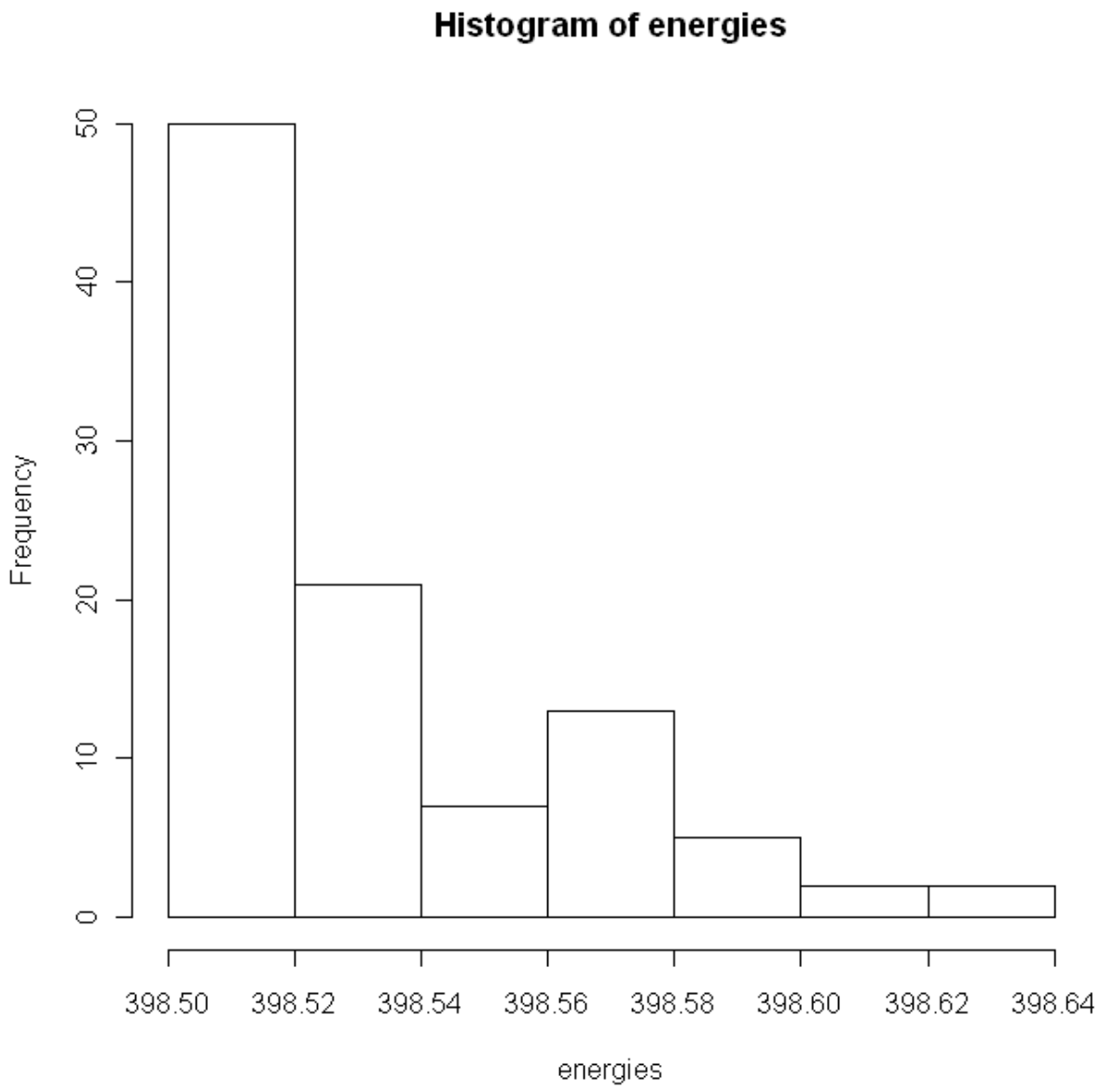
For the case where the wall charges are lower (here we are using point charges of 1.0 and wall charges of 10.0) there is a slight advantage to the square outside, single charge in the middle configuration. (V = 398.512 as opposed to 398.784.) A histogram of the energies of 100 different single annealer runs yields



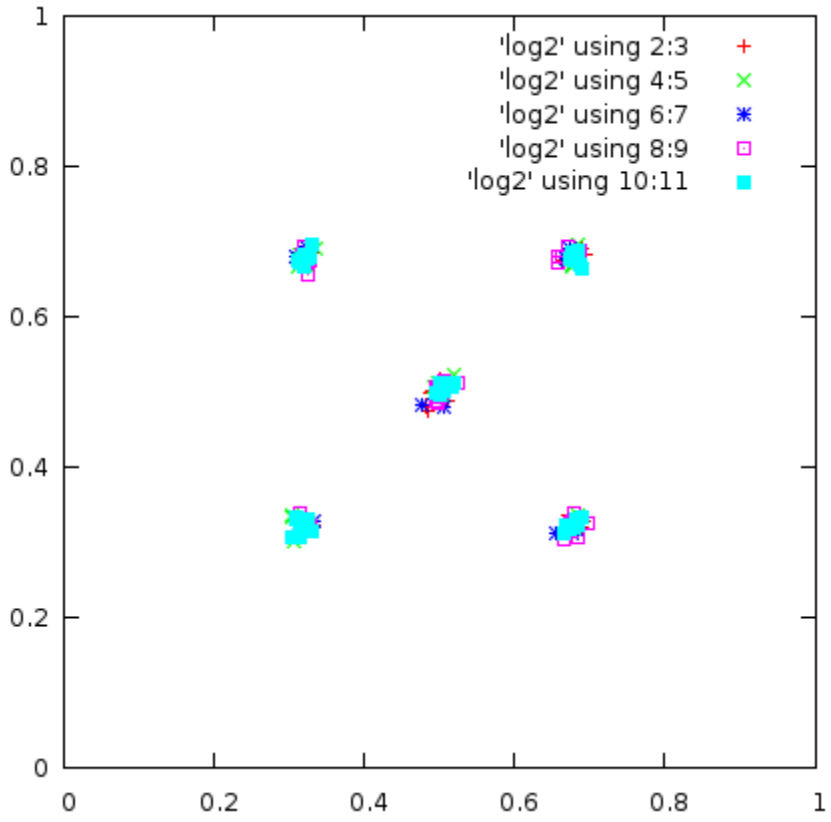
Note that roughly a 1/3 of the runs find the global minimum, with 2/3 getting stuck in the only slightly less preferable ring configuration. The 1/3 of the runs that find the optimum solution have the following configuration:



A drawback of the non-ensemble based simulated annealing method is that it does not lend itself to parallelization easily as the new position of the annealer depends on the previous position, creating a loop dependency. The ensemble based simulated annealing method, on the other hand, will allow itself to split over multiple annealers, each of which can be run concurrently. The results of the same histogram for 8 annealers are as follows:

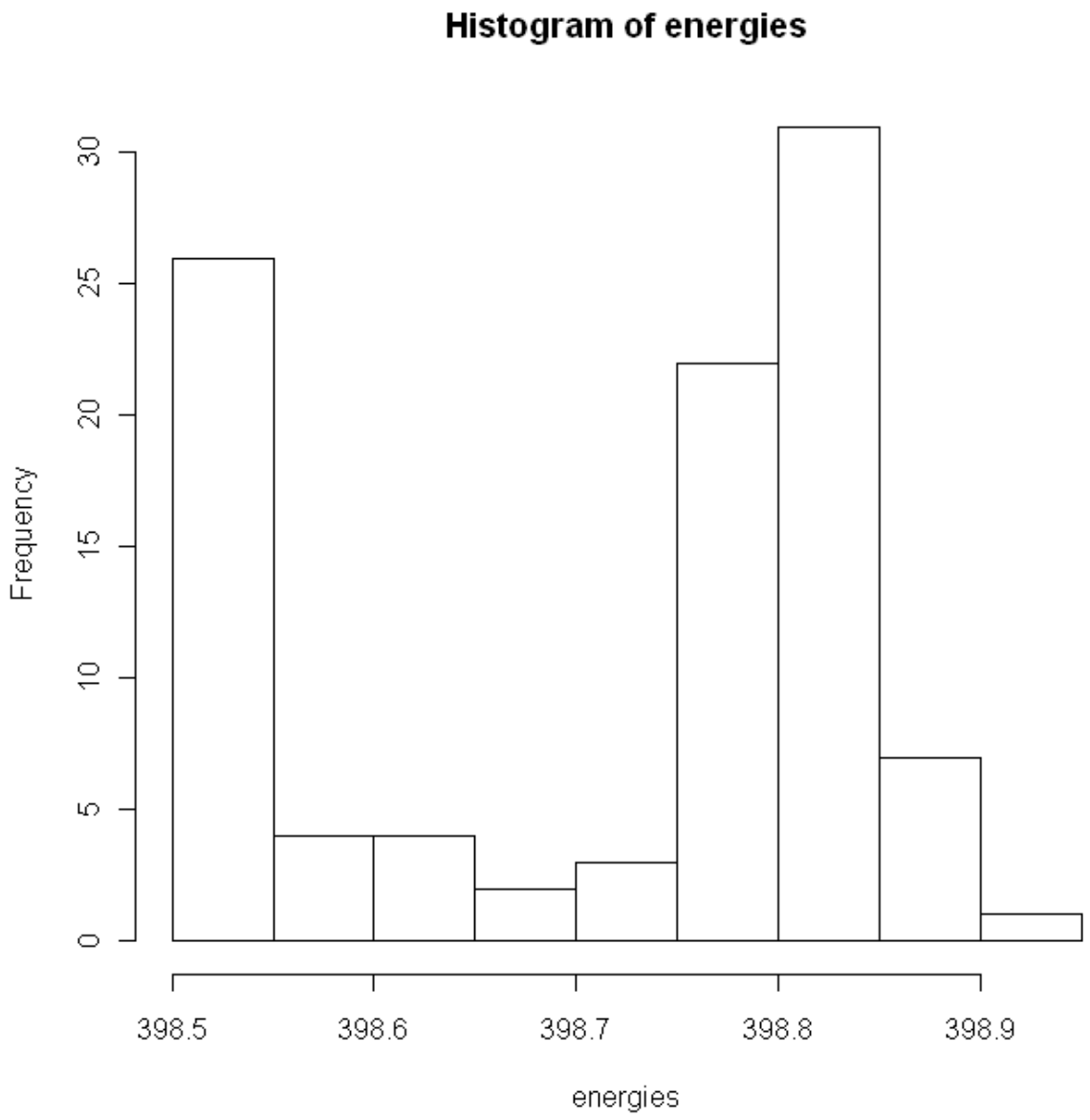


Note that while the ensemble based SA method gets the global minimum more often, it does not find it as precisely. See the spread in the solutions found in the ensemble method below.

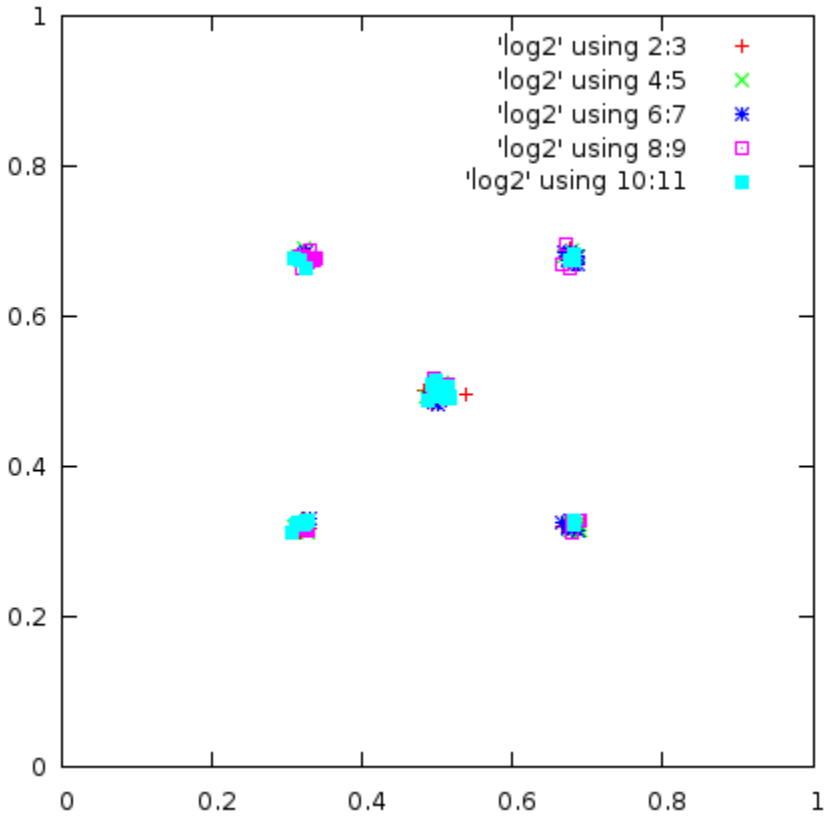


It would be interesting to see whether our other optimization methods find the same solution, and with what accuracy and precision.

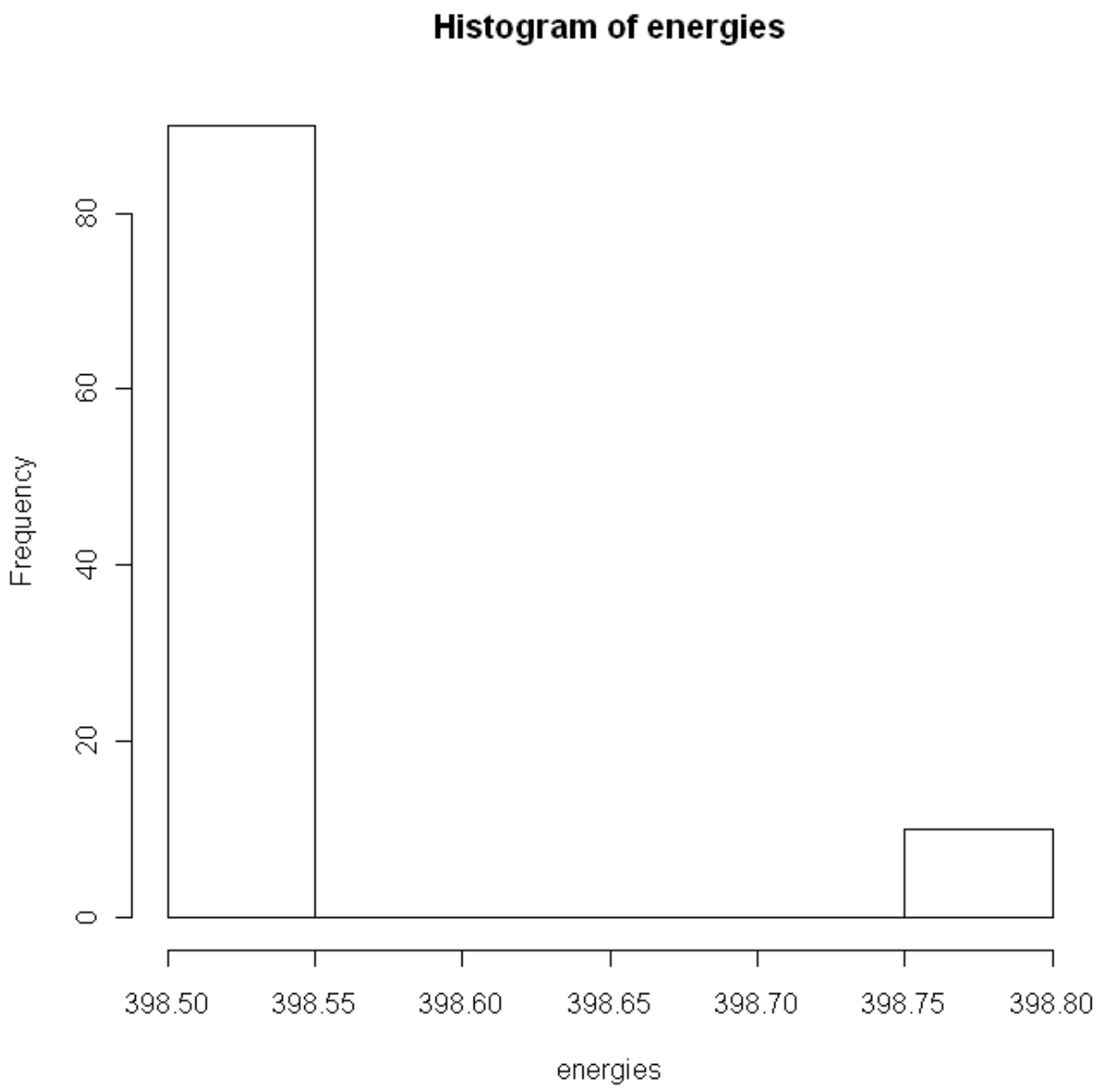
Using a “pooled” genetic algorithm, one gets the following histogram of energies from a 100-run ensemble:



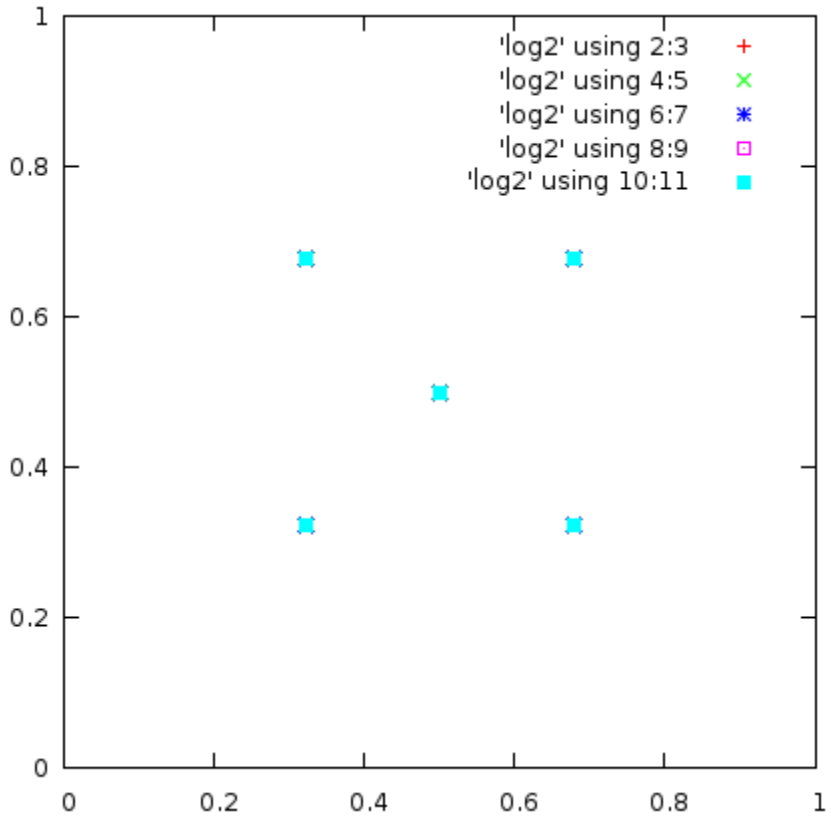
Looking at the spread in the optimum solution (only those with energies below 398.6 are shown below) one sees a similar result to the ensemble based simulated annealing in terms of precision—the genetic algorithm in this case lacked the precision of the single simulated annealing and lacked the accuracy of the ensemble based simulated annealing.



Finally, one might use a Monte Carlo – Powell's method approach to the same minimization, in this case choosing 5 random starting configurations for each Monte Carlo – Powell's optimization, and then comparing an ensemble of 100 Monte Carlo – Powell's runs.



Note that like the simulated annealing, each minimum found is found with great precision, but in this particular case, there is a very high success rate at finding the global minimum. Looking at the spread for the optimum solutions, we see the following configurations:



We see here the typical scenario when choosing optimization methods—a variety of methods that may or may not for a given problem truly find a global minimum, or that may or may not be able to determine degeneracies for near optimum solutions. In practice, you may find that for your particular optimization one method works better than another—your results will vary.

Exercises

1. Implement the OptLib 1.1.3 code to solve the optimization of the simsurface problem.
2. Increase the wall charge in the model. Is there a point where the geometric configuration of the 5 charge optimum solution changes from the box with a central charge? If so, at what wall charge?
3. Increase the number of charges in the model. What is the optimum solution for higher n, assuming that the ratio of the wall charge to the system charge stays the same (The default model was 40 charge units split between the 4 walls and 5 charge units for the particles—increase the wall charge to keep the proportions the same)?
4. For the genetic algorithm, run optimizations with different values of n_pop, n_keep, and n_pools. What do you notice about the effect each of these has on your convergence accuracy, precision, and number of total trials?

5. For the simulated annealing example, run optimizations with different initial temperature values. What do you notice about the effect on convergence accuracy, precision, and number of total trials?
6. For each method, make a scatter plot of multiple optimizations with different input parameters, comparing accuracy and precision for each method as a function of total number of function calls.

MODEL FITTING

Consider the case of fitting a known model to a given set of data. This can be reduced to a minimization problem provided that the problem can be posed in terms of a "goodness-of-fit" function. The most commonly used goodness-of-fit function is the chi-squared

$$\chi^2 = \frac{\Sigma\left(x_{i\,predicted} - x_{i\,observed}\right)^2}{\left|x_{i\,observed}\right|}$$

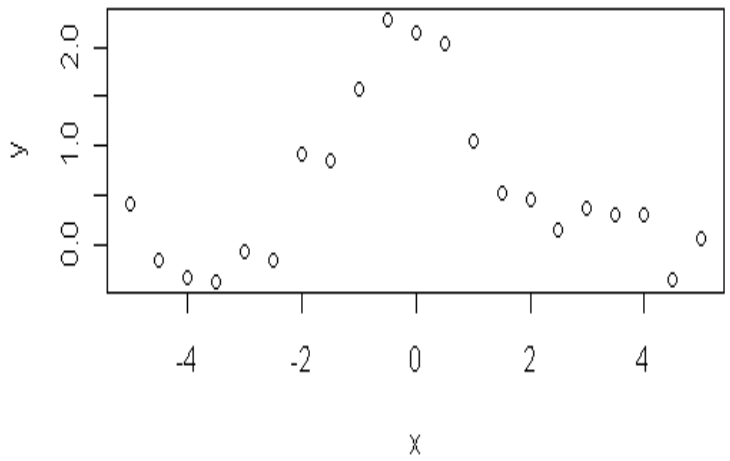
Note that larger values of x_i can dominate the chi-squared, and a modification often used in data fitting when the range of values which are important to fit span many orders of magnitude, (such as fitting the spectral energy distribution of a blackbody at many wavelengths) is

$$\chi^2 = \frac{\Sigma\left(x_{i\,predicted} - x_{i\,observed}\right)^2}{\left|x_{i\,observed}\right|^2}$$

Additionally, if all values are positive one might perform the chi-squared on the log of the values in the case where fitting many orders of magnitude is required.

Suppose our data to be fit is a single signal, with some background noise, that we suspect to be a Gaussian.

x	y
-5	0.41483
-4.5	-0.15759
-4	-0.33919
-3.5	-0.37158
-3	-0.0622
-2.5	-0.14863
-2	0.917062
-1.5	0.8596
-1	1.569149
-0.5	2.277346
0	2.142754
0.5	2.034874
1	1.042071
1.5	0.529893
2	0.45964
2.5	0.141449
3	0.37645
3.5	0.304982
4	0.307122
4.5	-0.35426
5	0.069823



A reasonable model fit to this would be $y = A\exp\left(-Bx^2\right)$, where A and B are chosen to minimize the chi-squared value between the model and the data.

Consider the following code from datafit.c, which reads in a text file of the above data (format is # of points on first line and then x,y pairs on successive lines) and performs an optimization.

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

#define MAX_LINE_LENGTH 240

// data structure used to pass data to chi_squared routine
typedef struct {
    int n;
    double * x;
    double * y;
} data;

double chi_squared(double *x, void * func_data) {
    double a=x[0];
    double b=x[1];
```

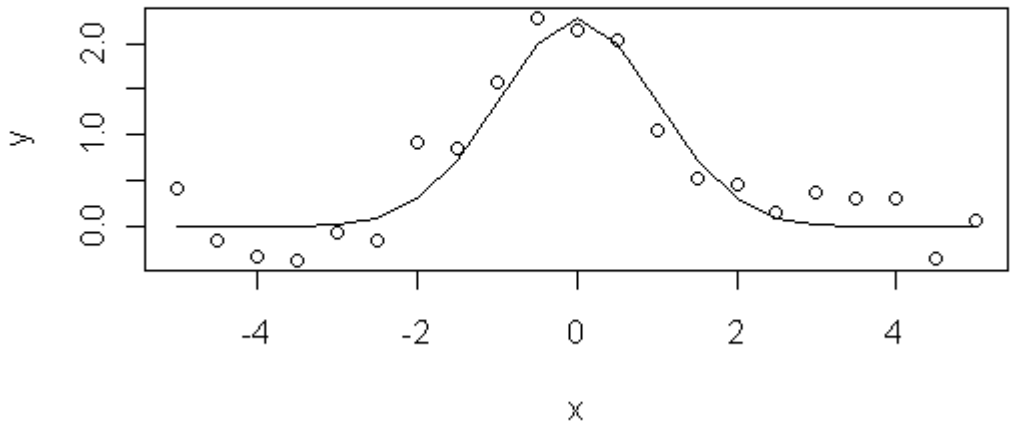
```
    data * theData = (data *)func_data;
    double sum;
    int i;

    // loop over data and calculate chi-squared assuming model
    //  of a*exp(-b*x*x)
    sum=0.0;
    for(i=0;i<theData->n;i++) {
        double f = a*exp(-b*theData->x[i]*theData->x[i]);
        sum += (f-theData->y[i])*(f-theData->y[i])/fabs(theData->y[i]);
    }

    return sum;


}

int main(int argc, char ** argv) {
    FILE * infile;
    char infile_name[MAX_LINE_LENGTH];
    char line[MAX_LINE_LENGTH];
    data theData;
    double guess[2];
    int i=0;


    // always seed your stochastic models
    seed_by_time(0);

    // read in data file
    // file format should be number of data items on first line,
    //   x,y, values on successive lines
    sprintf(infile_name,"data.txt");
    infile = fopen(infile_name,"r");
    theData.x=NULL;
    theData.y=NULL;
    theData.n=0;
    while(fgets(line,MAX_LINE_LENGTH,infile)!=NULL) {
        if(theData.n==0) {
            sscanf(line,"%d",&(theData.n));
            if(theData.n<1) {
                printf("ERROR: value of n=%d ");
                printf("not allowable\n",theData.n);
                exit(0);
            }
            theData.x = (double *)malloc(sizeof(double)*theData.n);
            theData.y = (double *)malloc(sizeof(double)*theData.n);
        } else {
            sscanf(line,"%lf %lf",&(theData.x[i]),&(theData.y[i]));
            i++;
        }
    }
    if(i!=theData.n) {
        printf("ERROR: value of i=%d not equal to n=%d\n",i,theData.n);
        exit(0);
    }
    fclose(infile);
```

```
    // inintialize guess
    guess[0]=1.0;
    guess[1]=1.0;

    // run OPTLIB_Minimize with defaults
    OPTLIB_Minimize(2,guess,&chi_squared,&theData,NULL);

    // output
    printf("SOLUTION a[%lf] b[%lf] chi-
squared[%lf]\n",guess[0],guess[1],
        chi_squared(guess,&theData));

    // free data
    if(theData.x!=NULL) free(theData.x);
    if(theData.y!=NULL) free(theData.y);

    return 0;
}
```

Here the function being minimized is the sum of square residuals normalized by the data values (i.e., the chi-squared value). Compiling and running the code should result in output similar to

```
SOLUTION a[2.265835] b[0.514572] chi-squared[3.918972]
```

which when plotted appears to fit the data:



Exercises

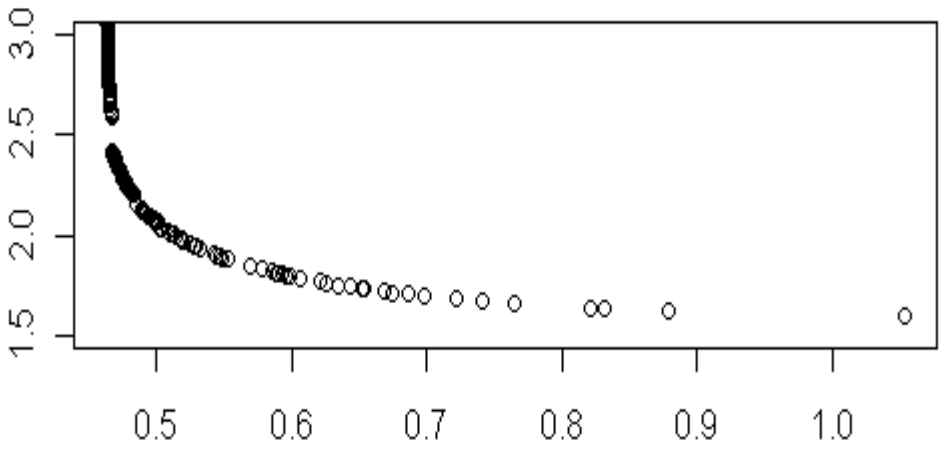
1. Modify the data-fitting code above to allow for a model of a Gaussian signal not centered at the origin with a linear background (see appendix B for sample datasets to fit).

2. Modify the data-fitting code above to allow for a model of a feature made up of two apparently Gaussian signals (see appendix B for sample datasets to fit).

MODEL FITTING WITH DEGENERACY

Consider the case of a model that will not have a single solution. A projectile is launched at some angle θ and some speed v towards a target that is 100 meters away and 20 meters in the air. There are multiple values of the inputs that will hit the target.

Running 1000 different optimizations (see code in Appendix C) results in the following ensemble of results (y axis is log(v), x axis is theta, units are SI) in which for many different angles at which the projectile is launched there is some speed that will hit the target—slower speeds for larger angles in which the projectile is lobbed and higher speeds at shallower angles in which the projectile heads straight to the target.



Note that this shows why the use of ensemble solutions is particularly important in stochastic minimization as it can bring out details of a degenerate solution—many solutions exist but the characteristic nature of the solutions and the range of the solutions are themselves worthy of study

Exercises

1. What is the range of angles for which a solution to the problem above is possible?
2. Can you explain the asymptotic behavior at low angles?
3. Note that the code in Appendix C includes penalties for guesses outside of a physically meaningful range, or in a range that is guaranteed to produce poor results. Could a transformation of variables be used so that a guess value on the real number line corresponds to the full range of acceptable values of v or theta (eliminating the need for penalties in the function being minimized)? Explain and present a transformation of variables that could be used

to ensure that a guess on the real number line corresponded to a physical value of v that was always positive. What transformation would ensure that a value on the real number line always corresponded to a value of theta between 0 and pi/2? Modify the code to implement.

PARALLEL PERFORMANCE

For each of these models, the parallel performance will depend on a few factors. The most important is the complexity of the function to be minimized—the less time that it takes to compute the function once the less likely it is that the parallel implementation will scale well. The reason for this is that the size of the problem in the traditional sense is not relevant to the number of function calls required to perform the optimization. A "bigger" problem being optimized will require more floating point operations per function call, and in all of the algorithms presented here the parallelization is performed at the level of a loop over function calls. More floating point operations per function call equates to more computation per communication as well as a greater degree of concurrent operation.

In the case of the MCP code, the parallel implementation assigns function calls to each process in a round-robin fashion in a SIMD algorithm. If you want to run 100 Monte Carlo trials and have 100 processors available, you could in principle run one trial per processor. The number of trials sets an absolute upper bound on the scalability of the problem, but if a single optimization completes too quickly, this may reduce efficiency further. The following shows wall time as a function of number of processors for a 1000 trial Monte Carlo—Powell minimization of a simple function call (finding the minimum of a parabolic 2-D well) compared to a simulated more complex problem, where the problem complexity is determined by the number of microseconds required to compute a function call, here implemented by adding a delay using the usleep command.

Number of Procs	1	2	4	8	16	32	64
Simple func call	0.166s	2.119s	1.099s	1.112s	2.597s	5.349s	5.761s
10 μs delay		354.862s	181.883s	95.492s	44.644s	30.244s	20.023s

While minimization of simple functions does not scale well, it does not in general need to as the minimization time is so short. However, even a moderately complex function call requiring 10 microseconds to complete shows significant scaling up to 64 processors for our MCP optimization.

In the case of ensemble-based simulated annealing (EBSA), parallelization is implemented when determining the step-size that allows for equilibrium at the given temperature. In the non-ensemble based algorithm, the annealer takes N_trials steps, each time keeping track of whether the step is accepted or not accepted according to the annealing rule, and the step size is adjusted up or down iteratively to bring the conditions into equilibrium. In the ensemble-based algorithm, multiple annealers divide up the N-trials steps, and the equilibrium state of the ensemble is determined rather than that of a single annealer. The maximum scaling in this case is limited by the number of annealers used, however there is also a point of diminishing returns in that more trials do not more efficiently determine equilibrium. In practice, setting N_trials to greater than 100-200 is unlikely to produce more accurate results, and the maximum possible scaling of EBSA is on the order of 100.

The GA algorithm proceeds in a generational process, with each new generation breeding the next. Each generation is ranked according to their ability to minimize the function, and the bulk of the computation lies in a loop over the population size N_pop evaluating f(x) for each. Maximum scaling then is tied to the population size. Like EBSA, there is a point of diminishing returns. Increasing the population size above some value does not increase the level of precision achieved nor does it decrease the number of generations required for convergence enough to warrant the greater size of N_pop. Values of N_pop of greater than 1000 are unlikely to produce results worth the additional computational effort, and thus the maximum scaling of GA algorithms is on the order of 1000.

In the case of both GA and EBSA, communication is required frequently (once per generation in GA and once per step size adjustment in EBSA). So, for maximum scaling to be achieved, a single function call should take considerably longer than the latency required for the communication.

In the case of both GA and EBSA, the stochastic nature of the solution is such that an additional ensemble approach must also be used—as one must be able to quantify the likelihood that the solution found deviates from the true global minimum. If local minima as well as the global minima are of interest (consider for example exploring the spectra of some molecule that could exist in a variety of meta-stable states, alternate binding sites of docking solutions of a ligand with a protein, or potential misfolded states of proteins), then additional detail would be desired in this ensemble, and instead of running GA or EBSA once, one might need to run EBSA 100-1000 times in order to perform appropriate statistics on the solutions found. Such parallelization might be programmed into a code directly, or done through submitting many identical jobs to a queue manager. Between the need for ensemble results and the maximum scaling of these algorithms, scaling from 10,000 to 1,000,000 is possible provided the function being minimized is sufficiently complex so that computation in the function offsets the overhead required to manage parallelization directly embedded in the code and in queue management.

Assuming a typical number of function calls required for a minimization of a function on the order of 100,000, and a need to run an ensemble of 100 to analyze the results of a stochastic optimization, the amount of work to be done in a real stochastic optimization problem is on the order of 10,000,000 function calls. In the case of a simple function (e.g., x^2+y^2 requires 2 multiplies and 1 add, which while technically 3 floating point operations will most likely benefit from some form of vector operation depending on chip and instruction set) it may very well be that the overhead of the function call consumes more CPU time than the function itself—in which case 10,000,000 operations on a CPU capable of 12,000,000,000 operations per second is not a problem requiring supercomputing capability.

On the other hand, if the function to be minimized requires a substantial amount of computation, such as a Monte Carlo or Finite Difference integration, or a more involved physical model, the problem may benefit from HPC. A function call requiring a tenth of a microsecond to compute (e.g. basic arithmetic expression plus function call overhead) could be minimized in this fashion in 1 second and parallel execution would incur more overhead than would benefit the problem. A function call requiring 10 microseconds (e.g. vector calculus operation on a 3-D grid) could be minimized in 100 seconds, which is long enough to benefit from parallel execution on a few cores. A function call requiring 1 second to execute (e.g. solution of small to moderate system of ODEs) would take about 100 days to complete on

a single core, and would significantly benefit from parallelism. A function call requiring 1 hour to execute (e.g. solution of large system of ODEs, PDEs, moderate N-body problem) would require 1000 years to be minimized in this fashion on a single core.

Exercises

1. Repeat the above example using as many processors as you can practically use on your cluster for the genetic algorithms and simulated annealing codes. How do the scaling properties of the different algorithms compare?
2. Repeat the above example for MCP, SA, and GA using the simsurface optimization. Compare for 5 charges, 25 charges, and 100 charges. Note that for the SA, you will need to change the number of annealers used in the code, and cannot use more processes than there are annealers, nor can you use more annealers than the trials per temperature in the code. For the MCP, you cannot use more processes than trials in the Monte Carlo optimization. For the GA, you cannot use more processes than the size of the population.

APPENDIX A: DEFINITION OF FUNC FOR SIMSURFACE PROBLEM

Spoiler alert, see below for code simsurface.c that could be used to define `func` for the simsurface problems.

```
#define HUGE 1.0e100
#define WALLCHARGE 10.0
#define POINTCHARGE 1.0
#define NOBJECTS 5

double func(double * x, void * func_data) {
    int * fcount;
    double retval;
    double wall_charge=WALLCHARGE;
    double point_charge=POINTCHARGE;
    int npoints=NOBJECTS;
    int i,j;
    double xi,yi,xj,yj,dx,dy,dr2,dr;
    double dt,db,dl;

    fcount = (int *) func_data;
    (*fcount)++;

    retval=0.0;
    for(i=0;i<npoints;i++) {
        xi = x[i*2];
        yi = x[i*2+1];
        // out of box penalty
        if(xi<=0.0) return HUGE;
        if(xi>=1.0) return HUGE;
        if(yi<=0.0) return HUGE;
        if(yi>=1.0) return HUGE;
        // other charges
        for(j=i+1;j<npoints;j++) {
            xj = x[j*2];
            yj = x[j*2+1];
            dx = xj-xi;
            dy = yj-yi;
            dr2 = dx*dx+dy*dy;
            dr = sqrt(dr2);
```

```
            retval += point_charge/dr;
        }
        dt = 1-yi;
        db = yi;
        dl = xi;
        dr = 1-xi;
        retval +=wall_charge*log((sqrt(dt*dt+dr*dr)+dr)/(sqrt(dt*dt+dl*dl)-dl));
        retval +=wall_charge*log((sqrt(db*db+dr*dr)+dr)/(sqrt(db*db+dl*dl)-dl));
        retval +=wall_charge*log((sqrt(dl*dl+db*db)+db)/(sqrt(dl*dl+dt*dt)-dt));
        retval +=wall_charge*log((sqrt(dr*dr+db*db)+db)/(sqrt(dr*dr+dt*dt)-dt));
    }
    return retval;
}
```

APPENDIX B: DATA-FITTING DATA SETS (NOTE: all data includes some noise)

y = 2*exp(-2.0*(x-1.0)^2)+ 0.4*(x) (data1.txt)

x	y
-5	-1.9928
-4.5	-1.81105
-4	-1.63309
-3.5	-1.39805
-3	-1.24602
-2.5	-0.99221
-2	-0.80842
-1.5	-0.63632
-1	-0.40487
-0.5	-0.20096
0	0.308276
0.5	1.446869
1	2.432385
1.5	1.857752
2	1.030071
2.5	1.05546
3	1.225137
3.5	1.402403
4	1.646815
4.5	1.827698
5	2.007179

y = 2*exp(-0.5*(x-1.0)^2/0.5)+ 1.0*exp(-0.25*(x+0.9)^2) (data2.txt)

x	y
-5	-0.09918
-4.5	0.167799
-4	0.208181

-3.5	0.25461
-3	0.366098
-2.5	0.581613
-2	0.822702
-1.5	0.914026
-1	1.135854
-0.5	1.085792
0	1.460454
0.5	2.22362
1	2.305738
1.5	1.672575
2	0.828277
2.5	0.289778
3	0.034668
3.5	-0.06035
4	0.109405
4.5	0.048599

APPENDIX C: PROJECTILE OPTIMIZATION PROBLEM (projectile.c)

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

// determination of vertical miss distance squared,
//  assuming target is 100 m away and 50 m up.
double target(double *guess, void * func_data) {
    double v = guess[0];
    double theta = guess[1];

    double x,y,vx,vy,t,dt;
    x = 0.0;
    y = 0.0;
    vx = v*cos(theta);
    vy = v*sin(theta);
    dt = 100.0/v/100.0;
```

```
    if(dt>0.01) dt=0.01;
    t = 0;
    if(theta<0.0||theta>M_PI/2) {
        return 10000.0+theta*theta;
    }
    if(vx<0.0) { // negative vx penalty
        return 10000.0+vx*vx;
    }
    while(x<100.0) {
        x += vx*dt;
        y += vy*dt;
        vy += -9.8*dt;
        t += dt;
        if(y<0.0) {
            return 10000.0+y*y; //landing too early penalty
        }
    }
    return (y-50.0)*(y-50.0); //squared to ensure minimum and
                              // not saddle point;
}

int main(int argc, char ** argv) {
    double guess[2];
    int i=0;
    int seed_offset=0;
    OptLibOpts * theOpts;

    // always seed your stochastic models
    if(argc>1) sscanf(argv[1],"%d",&seed_offset);
    seed_by_time(seed_offset);

    theOpts = OPTLIB_CreateOpts();
    printf("v theta e\n");
    for (i=0;i<1000;i++) {
        // inintialize guess
        guess[0]=drand(0,100.0);
        guess[1]=drand(0,M_PI/2.0);

        // run OPTLIB_Minimize with defaults
        theOpts->ga_itmax=200;
        theOpts->ga_itmin=10;
        theOpts->ga_dominancefactor=0.4;
        OPTLIB_Minimize(2,guess,&target,NULL,theOpts);

        // output
        printf("%lf %lf %lf\n",
            guess[0],guess[1],
            target(guess,NULL));
    }
    OPTLIB_DestroyOpts(theOpts);
```

```
    return 0;
}
```

APPENDIX D: SAMPLE ANSWERS TO EXERCISES

INTRODUCTION

Questions

1. What is the origin of the phrase "Monte Carlo" modeling? **The use of pseudorandom numbers to model stochastic phenomena, or to use random approximations to solve deterministic problems, is named after the city of Monte Carlo based on its history in the gambling industry.**
2. If optimization routines are typically set up to find the minimum of a function, what is the simplest way to modify a function in order to find the maximum instead? **Put a negative sign in front of the function, then minimize. The minimum of the opposite of a function is its maximum.**

AN EXAMPLE IN 2 VARIABLES

Exercises

1. Modify the function from Charbonneau to include any number of variables. Modify the func.c file included in the release of optlib 0.9 accordingly, and repeat the tests above for functions of higher numbers of variables. **There are any number of ways a student may attempt this. A simplistic approach might be to use a higher dimensional parabola, such as**

   $$f(\vec{x}) = \Sigma x_i^{\,2}$$

   A more direct implementation of the function as written would be

   $$f(\vec{x}) = 4^m \prod_{i=1}^{m} (x_i - 0.5)^2 \sin(n\pi x_i)$$

   Other variations might involve the product of a cosine and a decaying exponential in arbitrary dimensions

   $$f(\vec{x}) = \cos(n\pi r)\exp(-\gamma r^2) \qquad r = \sqrt{\Sigma x_i^{\,2}}$$

   Students should test against functions that have centers that are both zero and non-zero. To test against a non-zero minimum, the student can simply shift the function.

   $$f(\vec{x}) = \cos(n\pi r)\exp(-\gamma r^2) \qquad r = \sqrt{\Sigma (x_i - 0.5)^2}$$

2. How does the success rate (likelihood of an optimization attempt returning a correct global minimum) of each method change as the number of variables is increased?
   For the functions above, the number of iterations required for convergence may increase as the number of variables is increased, but adding in more free variables does not substantially change the ability of GA to find the global minimum. SA found substantially more local minimums, though it did still converge to the global minimum often enough to be used in an

ensemble approach. This could be corrected with a cooling factor closer to 1, but at a cost of additional iterations. For the well-constrained problem above, the Monte Carlo/Powell's hybrid did not appreciably differ between m=2 dimensions and m=8.

3. How does the total number of function calls required per optimization change for each method as the number of variables is increased? **GA required a small increase in number of iterations to converge at m=8 compared to m=4 and m=2. More dramatic increases are noticed at larger values of m. SA for a given cooling factor did not change the typical number of iterations required, but did require a higher cooling factor in order to converge consistently when the value of m was increased. The Monte Carlo/Powell's method hybrid did increase in the number of function calls per Powell's method step as m increased, and scaled as roughly m squared.**
4. For a function with a single global minimum and multiple nearby local minimums of a similar value (such as the function above or a combination of sinusoidal functions of high frequency and a Gaussian function with a large width) compare each individual method to a coupling of GA or SA with MCP in which GA/SA gets close to the global minimum and MCP "cleans up" the result, as well as to an approach in which GA/SA is used to get close and a single call of a deterministic method such as Powell's is used to clean up the result. Discuss the trade-off between total number of function evaluations required and likelihood of finding the global minimum. **A deterministic pass after a GA/SA run can effectively clean up the results, and does not suffer from the limitations that MCP does of having to a priori specify a range of values to be tested by the initial method. The primary choices that the student will have to make is the degree to which the convergence criteria on GA/SA can be loosened as well as whether to make a single deterministic pass or many using MCP. Loose convergence criteria in GA/SA coupled with a single deterministic pass can be useful at probing nearby local minima more effectively, strict convergence criteria coupled with MCP can improve accuracy in finding the global minimum. Students should have a method of determining the range of input values to be used in MCP based on GA/SA results. Students should also discuss whether running in parallel or serial would affect their choice of using Powell's method or Monte Carlo/Powell's to clean up GA/SA results.**

THE SIMSURFACE PROBLEM

Exercises

1. Modify the optlib 0.9 code to solve the optimization of the simsurface problem. **A solution to the function required is presented in Appendix A.**
2. Increase the wall charge in the model. Is there a point where the geometric configuration of the 5 charge optimum solution changes from the box with a central charge? If so, at what wall charge? **Increasing the wall charge will pack the particles more tightly and cause them to shift from a star configuration to a center packed square.**

3. Increase the number of charges in the model. What is the optimum solution for higher n, assuming that the ratio of the wall charge to the system charge stays the same (The default model was 40 charge units split between the 4 walls and 5 charge units for the particles—increase the wall charge to keep the proportions the same)? **Results will vary depending on how many charges the students choose. An interesting configuration is 7 charges, which will result in an "arrow" shaped minimum.**
4. For the genetic algorithm, run optimizations with different values of n_pop, n_keep, and n_pools. What do you notice about the effect each of these has on your convergence accuracy, precision, and number of total trials? **Results will vary, but typically students should see that for a given problem there is an optimum value of n_keep, generally between 5 and 15% of the total population size. A larger population can lead to better convergence up to a point, but for a given problem students should expect to see a point of diminishing returns between 100 and 1000. The use of tidal pools can result in minimizing the chance of finding a local minimum instead of the global minimum, but at a cost of more generations—however the average number of generations required to consistently find the global minimum at a given rate will generally be lower than running a single pool for a greater number of generations.**
5. For the simulated annealing example, run optimizations with different initial temperature values. What do you notice about the effect on convergence accuracy, precision, and number of total trials? **Starting with T too low can result in freezing into a local minimum. Starting with T too high can result in spending many iterations just readjusting the temperature. The latter can be helped by setting an appropriate maximum step size in the problem, as OptLib will adjust the temperature downwards if it cannot reach equilibrium without using a step size larger than the maximum step size.**
6. For each method, make a scatter plot of multiple optimizations with different input parameters, comparing accuracy and precision for each method as a function of total number of function calls. **Results will vary according to each student's particular minimization. For 4 charges, GA tends to converge in fewer iterations, but SA tends to find a slightly lower minimum. MCP did converge to the 4 charge solution, but with substantially greater function calls required. As the number of charges increased, SA continued to be more competitive at finding the global minimum, and also converges with fewer functions.**

MODEL FITTING

Exercises

1. Modify the data-fitting code above to allow for a model of a Gaussian signal not centered at the origin with a linear background (see appendix B for sample datasets to fit). **Students should include three additional parameters, one for the center of the Gaussian and two for the slope and intercept of the background, and modify the function accordingly.**
2. Modify the data-fitting code above to allow for a model of a feature made up of two apparently Gaussian signals (see appendix B for sample datasets to fit). **Students should include two additional parameters, for the amplitude and width of the second Gaussian, and modify the function accordingly.**

MODEL FITTING WITH DEGENERACY

Exercises

1. What are the range of angles for which a solution to the problem above is possible? **From the graph, the angle appears to range from 0.46 to 1.02.**
2. Can you explain the asymptotic behavior at low angles? **At a high enough speed, you simply aim for the target and gravity does not have enough time to affect the trajectory.**
3. Note that the code in Appendix C includes penalties for guesses outside of a physically meaningful range, or in a range that is guaranteed to produce poor results. Could a transformation of variables be used so that a guess value on the real number line corresponds to the full range of acceptable values of v or theta (eliminating the need for penalties in the function being minimized)? Explain and present a transformation of variables that could be used to ensure that a guess on the real number line corresponded to a physical value of v that was always positive. What transformation would ensure that a value on the real number line always corresponded to a value of theta between 0 and pi/2? Modify the code to implement. **An exponential/logarithm pair can be used to map the real number line to the positive number line. A tangent/inverse tangent pair can be used to map the real number line to a fixed range.**

PARALLEL PERFORMANCE

Exercises

1. Repeat the above example using as many processors as you can practically use on your cluster for the genetic algorithms and simulated annealing codes. How do the scaling properties of the different algorithms compare?
   Note: students may want to make comparisons both using wall time and using total number of function calls required, as for simpler problems where a single function call can be completed quickly scaling can often be very poor. Students can either modify the problem to a more involved one in which the function call takes longer to complete (adding in air resistance, using a smaller timestep, etc.) or add in an artificial delay (using the usleep command in unix for example) to investigate how a problem in which the function call takes longer to compute might scale. In general students will find that different problems scale differently with different methods, but that for each of these methods the primary issue determining the degree to which the problem scales will be the length of CPU time required for each function call—this particular function call completes very quickly, and in general the scaling of this problem will be poor.
2. Repeat the above example for MCP, SA, and GA using the simsurface optimization. Compare for 5 charges, 25 charges, and 100 charges. Note that for the SA, you will need to change the number of annealers used in the code, and cannot use more processes than there are annealers, nor can you use more annealers than the trials per temperature in the code. For the MCP, you

cannot use more processes than trials in the Monte Carlo optimization. For the GA, you cannot use more processes than the size of the population.

For the SimSurface problem, even with 100 objects students will most likely find that there is typically not enough work to be done to justify scaling beyond 8 processes, though there can be significant differences depending on the cluster architecture. Students should see for a very small number of charges GA and MCP are both very competitive with SA, but as the number of charges increases the number of GA generations and the number of function calls per Powell's step both increase. At a very large number of charges, MCP is inefficient for this problem and GA is competitive, but not as efficient as SA.