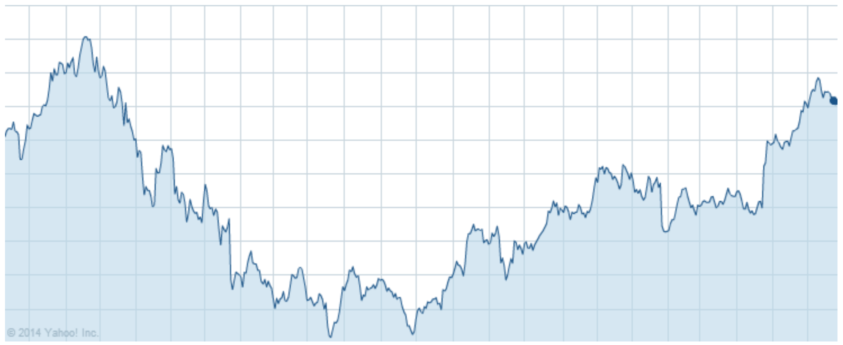
Stochastic Methods 2

- Simulated Annealing
- Genetic Algorithms

Simulated Annealing

Simulated Annealing is used for functions with many local extremes. Its goal is to return the global extreme as often as possible and, if not, then another almost-as-good extreme.

This is the type of graph for which simulated annealing might be appropriate:



Simulated Annealing

Simulated annealing begins with an initial point and tests neighboring points.

Unlike traditional optimization in which the neighboring point is immediately rejected if it is “worse” than the original, simulated annealing will accept some “worse” points in the beginning in an attempt to eventually find a global extreme.

Simulated Annealing

This link shows simulated annealing in action.
(credits to Wikipedia contributer Kingpin13)

The "temperature" measure indicates the willingness of the program to accept "worse" answers: high at first, lower later.

It's called "temperature" because the term *annealing* is borrowed from metallurgy where it describes a process of refinement by heating and then slowly cooling materials.

Simulated Annealing

If the algorithm for simulated annealing is run for long enough it will always find the global extreme. However it can sometimes be prohibitive to run that many calculations.

One of the advantages of simulated annealing is that even if you stop it a little early, its current solution will either be the optimum or really close to it.

Simulated Annealing

Here is a section of *pseudocode* showing the simulated annealing procedure for minimization:

```
s ← s0; e ← E(s)                        // Initial state energy.
sbest ← s; ebest ← e                    // Initial "best" solution
k ← 0                                   // Energy evaluation count.
while k < kmax and e > emax             // While time left & not good enough:
  T ← temperature(k/kmax)               // Temperature calculation.
  snew ← neighbour(s)                   // Pick some neighbour.
  enew ← E(snew)                        // Compute its energy.
  if P(e enew T) > random() then      // Should we move to it?
    s ← snew; e ← enew                  // Yes change state.
  if enew < ebest then                  // Is this a new best?
    sbest ← snew; ebest ← enew          // Save 'new neighbour' to 'best found'.
  k ← k + 1                             // One more evaluation done
return sbest                            // Return the best solution found.
```

(again credits to Wikipedia)

Practice Problem 1

1. Translate as much of this pseudocode as possible into proper Julia language:

```
s ← s0; e ← E(s)                                // Initial state energy.
sbest ← s; ebest ← e                            // Initial "best" solution
k ← 0                                           // Energy evaluation count.
while k < kmax and e > emax                     // While time left & not good enough:
  T ← temperature(k/kmax)                       // Temperature calculation.
  snew ← neighbour(s)                           // Pick some neighbour.
  enew ← E(snew)                                // Compute its energy.
  if P(e enew T) > random() then              // Should we move to it?
    s ← snew; e ← enew                          // Yes change state.
  if enew < ebest then                          // Is this a new best?
    sbest ← snew; ebest ← enew                  // Save 'new neighbour' to 'best found'.
  k ← k + 1                                     // One more evaluation done
return sbest                                    // Return the best solution found.
```

Genetic Algorithms

Genetic algorithms attempt to find optimal solutions by using concepts related to the process of natural selection:

inheritance

mutation

selection

and *crossover*.

Building a Population

Each “individual” is represented by a string, usually in binary; for example:

1111100010100111011010001011011001011011

Fortunately any number can be converted into binary and back so this string of 0’s and 1’s can be translated into the values of numerical variables. These values can then be plugged into an objective function.

In the beginning a large number of these strings are randomly chosen.

Testing Fitness

The next step is to find the value of the objective function for each “individual”. This is done by translating the binary strings into numbers and plugging them in. The objective function value gives a measure of the individual’s “fitness”.

If you are trying to minimize, the “fittest” individuals would have the lowest values of the objective function.

Binary Numbers: Review

With binary numbers, each digit space represents a power of two. This number:

1	0	0	1	1
16’s	8’s	4’s	2’s	1’s

…contains 1 sixteen, 0 eights, 0 fours, 1 two and 1 one. That adds up to 16 + 2 + 1 = 19.

Similarly, 101 = ?

5

1010 = ?

10

Practice Problem 2

Here are four randomly-generated individuals:

A: 011111010001 B: 001000101001

C: 101110001010 D: 101101000111

2a. Split each "phenotype" into three variables of length 4. Convert each into a base-10 number, represented by a, b, and c.

2b. For each individual, calculate its "fitness" according to the objective function a + b – c, with the goal of *maximizing*.

Selection

After fitness is determined, the individuals are ranked from most to least fit. All of the individuals may contribute to the next "generation", but with different probabilities based on their fitness.

Individuals are chosen randomly, in pairs, with repeats allowed, until the number chosen is equal to the original population.

Practice Problem 3

In Problem 2, the fitness in order from high to low was A, C, D, B.

Randomly choose pairs of individuals according to the following method:

Use the command `rand(1:10)`. If the result is 1, 2, 3, 4 choose A; 5, 6, 7 choose C; 8, 9 choose D; 10 choose B.

Individuals may not “breed” with themselves; choose another number if this happens.

Use this method to create two pairs.

Crossover

In the next step, a random number *n* is selected between 1 and one less than the maximum string length. Then, the first n characters of each pair are switched.

If you had the pair 10110 / 01101 with n = 3

you would switch the first three characters of each

creating two "children," 01110 / 10101 .

Practice Problem 4a

For each pair from problem 3, use `rand(1:11)` to determine how many digits to swap; then, swap those digits.

Mutation

The process of “mutation” is a random process that prevents the “children” from being too similar to the “parents”, which would make it difficult to break out of a local optimum and find the global one.

In mutation, a probability is selected; it can be a chosen number between 0.005 and 0.1, or equal to 1/k where k is the string length (the second method guarantees an average of 1 mutation per child).

Each character is given that probability of randomly switching value from 0 to 1 or 1 to 0.

Practice Problem 4b

Use the command `rand(12)` to generate 12 random numbers from 0 to 1.

If any of the numbers is less than $0.08\overline{3}$ (which happens to be 1/12), flip the corresponding character(s) on the first child from 4a.

Repeat for all four children.

Calculate the fitness of all four children.

The Last Step

Using computers, these procedures are repeated over large numbers of individuals over many generations. Eventually, the “population” will become more “fit”, just as natural selection works on a biological population.

In more mathematical terms, this means the variables and fitness will gradually converge on the optimal solution.

Practice Problem 5

Repeat the genetic algorithm for two more generations: pair the children with weighted probabilities according to fitness, swap genes, mutate, evaluate; then repeat once more.

Genetic Algorithms

One disadvantage of genetic algorithms is that they require a lot of evaluations of the objective function, which can be costly and time-consuming.

Another is that, while they work well for simpler systems, they do not scale up well to very complex systems: the longer the string, the less successful the method tends to be.

Genetic algorithms also tend to get stuck in local extremes.

Genetic Algorithms

In spite of these difficulties, for simple systems (fewer variables) with many potential inputs (large domains) and complicated, interdependent, or nonexistent equations and conditions, genetic algorithms are surprisingly good at finding optimal solutions.