Monte Carlo Simulation.[*]

Greg Kochanski
http://kochanski.org/gpk

2005/03/10 10:45:56 UTC

1 Introduction

The idea behind Monte-Carlo simulations gained its name and its first major use in 1944 [Pllana, 2000], in the research work to develop the first atomic bomb. The scientists working on the Manhattan Project had intractably difficult equations to solve in order to calculate the probability with which a neutron from one fissioning Uranium[1] atom would cause another to fission. The equations were complicated because they had to mirror the complicated geometry of the actual bomb, and the answer had to be right because, if the first test failed, it would be months before there was enough Uranium for another attempt.

They solved the problem with the realization that they could follow the trajectories of individual neutrons, one at a time, using teams of humans implementing the calculation with mechanical calculators [Feynman, 1985, Man, 2004]. At each step, they could compute the probabilities that a neutron was absorbed, that it escaped from the bomb, or it started another fission reaction. They would pick random numbers, and, with the appropriate probabilities at each step, stop their simulated neutron or start new chains from the fission reaction.

The brilliant insight was that the simulated trajectories would have identical statistical properties to the real neutron trajectories, so that you could compute reliable answers for the important question, which was the probability that a neutron would cause another fission reaction. All you had to do was simulate enough trajectories.

When Simulation is Valuable.: Q: In a free fall, how long would it take to reach the ground from a height of 1,000 feet? **A:** I have never performed this experiment.

2 Simple Example

2.1 Birthday Problem - Classical Approach

Simple examples of Monte-Carlo simulation are almost embarrassingly simple. Suppose we want to find out the probability that, out of a group of thirty people, two people share a birthday. It's a classic problem in probability, with a surprisingly large answer.

Classically, you approach it like this: Pick people (and their birthdays) randomly, one at a time. We will keep track of the probability that there are no shared birthdays.

- The first person can have any birthday, and there is still a 100% chance of no shared birthdays.
- The second person has one chance of overlapping with the first person, so there is a 364/365 chance of placing him/her without an overlap. The probability of no shared birthdays is 364/365.

[1] Or, Plutonium, of course.

- The third person has two chances of overlapping with the first two people, so there is a 363/365 chance of placing him/her without overlaps (two days are taken). The probability of no shared birthdays is now $(364/365) \cdot (363/365)$.
- The fourth person has three chances of overlapping with the first three people, so there is a 362/365 chance of placing him/her without overlaps. The probability of no shared birthdays is now $(364/365) \cdot (363/365) \cdot (362/365)$.
- ...
- The thirtieth person has 29 chances of overlapping with the first three people, so there is a 336/365 chance of placing him/her without overlaps. The probability of having no shared birthdays is now $(364/365) \cdot (363/365) \cdot (362/365) \cdot \ldots \cdot (336/365)$.

The overall probability of no overlapping birthdays is then 0.294, giving a 71% chance that at least one pair of people have overlapping birthdays. It's not too complex if you see the trick of keeping track of the probability of zero overlaps, rather than trying to add up the probability of one or more overlaps. It also takes some thought to realize that the probabilities are conditioned properly, so that multiplying together all the various $P(N^{\text{th}}$ person doesn't overlap$|$first $N-1$ people don't overlap) factors.

2.2 Birthday Problem – Monte-Carlo Approach

The solution here is conceptually very simple:

1. Pick 30 random numbers in the range [1,365]. Each number represents one day of the year.
2. Check to see if any of the thirty are equal.
3. Go back to step 1 and repeat 10,000 times.
4. Report the fraction of trials that have matching birthdays.

A computer program in Python to do this calculation is quite simple:

```
#!/usr/bin/env python

import random                  # Get a random number generator.

NTRIALS = 10000                # Enough trials to get an reasonably accurate answer.
NPEOPLE = 30                   # How many people in the group?

matches = 0                    # Keep track of how many trials have matching birthdays.
for trial in range(NTRIALS):                    # Do a bunch of trials...
        taken = {}                              # A place to keep track of which birthdays
                                                #         are already taken on this trial.

        for person in range(NPEOPLE):           # Put the people's birthdays down, one at a time...
                day = random.randint(0, 365)            # On a randomly chosen day.
                if day in taken:
                        matches += 1                    # A match!
                        break                           # No need to look for more than one.
                taken[day] = 1                          # Mark the day as taken.

print 'The fraction of trials that have matching birthdays is', float(matches)/NTRIALS
```

And the answer is:

```
The fraction of trials that have matching birthdays is 0.7129
```

3 Example in Class

- How many raisins do you add to a batch of dough to make M cookies to make sure (with probability P) that a random cookie has at least N raisins?
- How about that 99% of the cookies will have at least one raisin?
- How about that all the cookies will (with probability P) have at least one raisin?

4 A Linguistic Example

Let's try an artificial model with a certain amount of linguistic reality, to see how Monte-Carlo techniques might be applied to bigger problems.

Imagine we are studying English names, and wish to understand their origins. We hypothesise that names were generated by the operation of five processes:

1. A name could be prepended by a random descriptive nickname: (*e.g.,* "speedy").
2. A name could be lengthened by appending a place name or an occupation.
3. A name could be shortened by dropping any sequence of adjacent syllables, so long as two syllables remain.
4. If a new name is identical to a place name, reject the change that led to this and try again.
5. If a new name is identical to a common name, possibly reject the change that led to this and try again.

How can we test this hypothesis? How can we even understand what it will do? How can we compare it to recorded names?

We will build a little universe that generates names, and then compare the simulated names to the real records that we have. Specifically, we will run the simulation to produce a good-sized sample of data, then compute $P(\text{Data}|\text{Model}_i)$. We can do this several times with several different models and compare them by using (you guessed it!) Bayes' Theorem to compute $P(\text{Model}_i|\text{Data})$. We can also do various standard hypothesis tests to see how well any of our attempted models match the data. Thus, we can test the model by simulating it, and seeing how well it reproduces the names that we have collected.

Now, the above description of the hypothesis is not complete. It is the framework for a family of related models (models in the sense of Bayes' Theorem). In order to work with this, we need to make this hypothesis specific enough so that we can actually compute one or more lists of names to compare to reality. To complete the hypothesis, we need to know how often these five processes operate. We need to know things like a probability distribution for the nicknames in process 1. We need to know the distribution of place names or occupations in process 2. We need to know the probability distribution of the number of syllables to drop in process 3.

Some of these things we can guess from existing documents: lists of place names and occupations, along with their relative frequencies. Others, we can't guess, not without looking at our data, the names in the documents. Consequently, we don't guess; we will put an adjustable parameter in the model for these numbers, and we will

What is an adjustable parameter?: An adjustable parameter is a place-holder in a family of models. Every value of the parameter (or each combination of the parameters, if you have more than one) creates a new model. Often, you look for the value(s) of the adjustable parameter(s) that give the best fit to some data.

eventually let the combination of the data and model tell us which which values for the adjustable parameters work and which don't.

4.1 How do we compare the simulation to reality?

Given that there are so many possible names, it might be too much to ask our little simulation to produce the exact set of names that exist in the documents. Not that it would be intrinsically wrong to ask the simulation to produce real names, but you could easily run into either of two practical problems:

1. You might have to run far too many simulations before you find one that produces more than a few of the names that actual Englishmen had. It could well be that the set of possible names was far larger than the set of Englishmen, for instance. If so, even a correct simulation might well produce the names of Englishmen in some alternate universe – names that they might have had, had their mothers married someone different. This could mean that you run out of time or money before you find a good solution.

2. You might find that you need to know too many things to get the solution to match in enough detail. Suppose you needed to know the probability that each possible nickname could end up in a name? One cannot deduce that information without looking at some data; if looking at descriptive nicknames in other languages is not sufficient, your only hope would be to put an adjustable parameter in for the probability of each nickname, and hope that the data will actually allow you to choose these values.

   By doing that, you will probably end up with a family of models that has too many adjustable parameters for the available data. Aside from taking longer and costing more, you will find that many of these models in the family are equally good fits to the available data, and consequently, there are many possible combinations of adjustable parameters that are equally likely, given the data.

Degeneracy: The technical term for a situation where the data allow many (nearly-) equally good hypotheses is "degeneracy". A family of models is degenerate with respect to a set of data if the data are not sufficient to select a single model (or a small group of closely related models) out of the family.

   The real problem is that, along with many possible values of these parameters that you might not really care abut, any adjustable parameters that you actually *do* care about may well become quite uncertain, too.

Consequently, it might be more practical, rather than matching names in detail, to match classes of names, and to see how well the simulation generates names in the right class, even if the names are not exactly the same. For our example here, our classes will be names of one, two, three, ... syllables, and names that contain the syllable "smith".

4.2 Implementing the Model

The complete model, as implemented, assumes that the probability of process 3 (shortening) is proportional to length of the name, and that the probability of process 5 (rejection) is proportional to the frequency of the name you bump into. We assume that nicknames, place-names, and occupations are simply drawn from an equally-probable list. (Needless to say, this shouldn't be taken too seriously as an actual model of name generation.)

Thus, there are four adjustable parameters: the probability that processes 1 and 2 will operate on a name in each generation, and the constants of proportionality for processes 5 and 3. We initialise the names from place and occupational names[2]. I run the model for a constant population of 10,000 people, for 20 generations.

Now, with the parameters set to (0.25, 0.25, 0.25, and 5), the simulation runs in 30 seconds, and yields the following names (these are the most common names, each followed by their frequency):

[2] Initialising the names from place and occupational names was an an arbitrary choice, which was made on the assumption that it wouldn't matter. That is indeed the case with the set of adjustable parameters that I started with. With the initial set of parameters, most names would be heavily modified within 20 generations: syllables would be added and deleted, and the initial name would typically have been removed at one generation or another. The processes (with the initial values) would lose all memory of the original names fairly rapidly. However, as it turns out, the best-fit parameters that we will derive later do not modify the names so rapidly. If this were a real analysis instead of a tutorial, we'd have to go back and think carefully about what the initial set of names might be.

Name	Occurrences	Name	Occurrences	Name	Occurrences	Name	Occurrences
far-rier	49	butch-er	49	black-smith	49	sai-er	47
por-ter	45	weav-er	44	farm-er	43	gold-smith	42
mer-town	41	ding-ton	40	bar-ber	39	push-er	38
ling-ton	37	scho-lar	36	red-er	36	brew-er	36
goat-herd	35	shep-herd	34	tan-er	33	con-er	33
book-er	32	ba-ker	32	spee-ter	31	cow-herd	31
tai-lor	30	cow-man	30	stink-er	29	spee-er	29
bur-ry	29	black-ter	29	black-er	29	big-er	29
pen-ter	28	ing-don	27	arch-er	27	scot-er	26
red-ter	25	mas-ter	25	stink-smith	21	...	...

The names have some relation to recent English names, but there are lots of names that simply don't often occur in the real world (*e.g.* "penter", "stinksmith", "speeter"). And, finally, because it is built from a small number of occupational names, place names, and nicknames, the simulation will not cover all possible syllables, so it misses many English names. Some of these failings are intrinsic to the family of models; some could be improved by adjusting the adjustable parameters, and some could be improved by getting a complete set of place names[3]. However, we anticipated some of these problems, and we are not making a word-by-word comparison. We are going to see if it gets some of the statistical properties of names correct: specifically the distribution of lengths and the fraction of names that contain "smith".

We will compare it to statistics from a list of 27882 names of New Jersey employees of Lucent Technologies. Of these, 27041 had given names, rather than an initial in the database. We will approximate a syllable count for each name by assuming that you get one syllable per three letters, and rounding to the nearest integer.

In all, 10.6% of the names that the simulation produces contain "smith", compared to 0.6% in our reference data. Looking at the length histogram, we get the following distribution of lengths:

Length	Fraction simulation	Fraction data
1	0	0
2	43%	3.6%
3	20%	20%
4	12%	46%
5	10%	23%
6	7%	5%
7	4%	0.9%
8	2%	0.2%
9	1%	0.02%
...	...	

No single syllable names.: The fact we generate no one-syllable names is interesting. That lack comes primarily from process 3, which never shortens a name down to one syllable, combined with processes 1 and 2, which will, over the course of many generations, extend single-syllable names by either adding a prefix nickname or a place or occupation name as a suffix.

As you can see, this isn't a very good match. Our simulation gives far too many Smiths, far too many two-syllable names, and too many names with seven or more syllables. If we do a hypothesis test, comparing our statistics, using (for instance) a chi-squared test on the difference between the simulated results and the actual data, we will find that the we can reject this model at the 99.99% confidence level or beyond.

However, this is not the only model in the family. We have four adjustable parameters we shall adjust, in hopes of getting a better representation of the data.

[3] And, we should probably weight place names by their population. After all, there were a lot more people from York than from Wytham.

MAP and Hypothesis testing: Note that you may want to know which is the best of N different models. That would involve a MAP (Maximum A-Posteriori) comparison between models. This question of best-fit is different from asking whether a particular model is acceptable by a hypothesis test. While one prefers a situation where the best fit model of a family is not rejected, it happens sometimes. If so, it means that the family is not an accurate description of the data. Even so, the best fit model could still be interesting either as an approximation to the data or as an important clue toward a future, better description of the data.

If one changes the adjustable parameters to improve the match between the simulation and the data, one can obtain a considerably better match to the data, as you can see from the table to the left. We still have far too many Smiths (8% vs. 0.6%), but most of the frequencies of lengths of simulated names are now within a factor of two of the data. Overall, it is still not a good fit to the data, even though we have reduced the chi-squared statistic by a factor of three.

Length	Fraction **best-fit** simulation	Fraction **data**	Fraction first simulation
1	0	0	0
2	19%	3.6%	43%
3	31%	20%	20%
4	27%	46%	12%
5	14%	23%	10%
6	6%	5%	7%
7	2%	0.9%	4%
8	0.5%	0.2%	2%
9	0.2%	0.02%	1%
...	...		

A sample of the most common names produced by this best-fit model (within its family) follows. Some of the names look quite plausible:

Name	Occurrences	Name	Occurrences	Name	Occurrences	Name	Occurrences
scot-er	29	blond-er	26	shrimp-er	24	ee-er	24
red-er	22	push-ter	22	stink-ee-er	21	big-er	21
spee-er	20	book-worm-er	20	book-er	19	shrimp-ton	18
shrimp-smith	18	spee-dee-ter	17	red-smith	17	red-ee	17
big-ter	17	stink-ee	16	shrimp-ter	16	ee-smith	16
black-er	16	stink-ter	15	stink-smith	14	push-ee-er	14
black-smith	14	stink-don	13	spee-ter	13	spee-smith	13
sai-ee	13	book-ter	13	black-ter	13	big-don	13
spee-ley	12	scot-ton	12	scot-smith	12	scot-ee	12
sai-lor-er	12	dee-er	12	blond-ee-er	12	big-smith	12
stink-ee-ee-er	11	shrimp-herd	11	sai-ter	11	sai-lor-ter	11
sai-ley	11	ee-ter	11	...	...	...	...

So, what did we do to the adjustable parameters to improve things?

- We increased the probability of the first process (prepending a nickname) from 25% to 44% per generation.
- We reduced the probability of appending a place or occupational name from 25% to 0.01%. To all practical purposes, we have completely ceased to include place and occupational signifiers into names. Any occupational names that persist (*e.g.* "Smith") are survivals from the initial set of names.
- We have reduced the probability of syllable deletion: it is now 0.14 times the length of the name, rather than 0.25 times the length of the name.
- Finally, we have dramatically reduced the extent to which new names avoid existing names. To a good approximation, a name is now generated without consideration of whether or not other people have the same name.

If this were a real paper, or if the fit were better, we would conclude that to the extent that this model is valid, names last for many generations, and that nicknames have recently been[4] more important as a source of

[4] Over the last 10 or 20 generations.

syllables than occupational or place names. However, this is just a toy model, and should not be taken seriously, except to point out a path that could be followed.

The Python source code for the relevant parts of the model is found in Appendix A.

References

Richard Feynman. *Surely You're Joking Mr. Feynman!* Bantam, 1985.

History – Los Alamos – Oversight Committee Formed. The Manhattan Project Heritage Preservation Association, http://www.childrenofthemanhattanproject.org/HISTORY/H-06c12.htm, 2004. URL `http://www.childrenofthemanhattanproject.org/HISTORY/H-06c12.htm`.

Sabri Pllana. History of Monte Carlo method. http://www.geocities.com/CollegePark/Quad/2435/index.html, August 2000. URL `http://www.geocities.com/CollegePark/Quad/2435/index.html`.

A Computer Implementation of the Model

```
#!/usr/bin/env python

"""This script simulates the generation of English names.
It is also used to find the set of parameters that does
the best job of generating names.

This work is available under http://kochanski.org/gpk/teaching/0401Oxford ,
part of the lecture titled ``Monte Carlo Simulations,''
from the Hilary Term 2004 course.
"""

# This work is licensed under the Creative Commons Attribution License.
# To view a copy of this license,
# visit http://creativecommons.org/licenses/by/1.0/
# or send a letter to Creative Commons,
# 559 Nathan Abbott Way, Stanford, California 94305, USA.
#       HISTORY
#           Written and copyright by Greg Kochanski, 2004.


import random               # Random number generators.
import Numeric              # Math on vectors and matrices.
import math                 # Other maths functions.

# A list of nicknames.  All are assumed to be equally probable.
Nicknames = [
                ['red'],
                ['spee','dee'],
                ['big'],
                ['push', 'ee'],
                ['blond', 'ee'],
                ['shrimp'],
                ['stink', 'ee'],
                ['book', 'worm'],
                ['sai', 'lor'],
                ['scot'],
                ['black']
                ]


# A list of occupational names:
Occ = [
                ['smith'], ['butch', 'er'], ['farm', 'er'],
                ['cow', 'man'], ['weav', 'er'],
                ['tai', 'lor'], ['tan', 'er'],
                ['brew', 'er'], ['vel', 'lum', 'mak', 'er'],
                ['car', 'pen', 'ter'], ['groom'],
                ['far', 'rier'], ['black', 'smith'],
                ['bar', 'ber'], ['gold', 'smith'],
                ['arch', 'er'], ['cook'], ['ba', 'ker'],
                ['cow', 'herd'], ['shep', 'herd'],
                ['goat', 'herd'], ['fal', 'con', 'er'],
                ['scho', 'lar'], ['mas', 'ter'],
                ['por', 'ter']
                ]

# A list of place names:
Place = [
                ['ox', 'ford'], ['hink', 'sey'],
                ['wy', 'tham'], ['thame'],
                ['wynch', 'wood'], ['bot', 'ley'],
                ['sum', 'mer', 'town'],
                ['lon', 'don'], ['york'],
                ['ches', 'ter'], ['read', 'ing'],
                ['bath'], ['ave', 'bur', 'ry'],
                ['dor', 'ches', 'ter'], ['mar', 'ston'],
                ['hea', 'ding', 'ton'], ['cow', 'ley'],
                ['cum', 'nor'], ['kid', 'ling', 'ton'],
                ['saint', 'giles'],
                ['ab', 'ing', 'don']
        ]


class ModelParameters:
        __doc__ = """This class contains the adjustable parameters
                        for the family of models."""

        def __init__(self, prms=None):
                """This function creates an instance of the class."""
                if prms is None:                # Default parameters
                        self.p1 = 0.25
                        self.p2 = 0.25
                        self.p3 = 0.25
                        self.pdup = 5.0
                else:
                        # Take parameters from an array on the argument list.
                        self.p1, self.p2, self.p3, self.pdup = prms

        def not_ok(self):
                """This function tests if the adjustable parameters are silly
                or not.
                """
                if self.p1<0 or self.p1>1:
                        return 'p1'
                if self.p2<0 or self.p2>1:
                        return 'p2'
                if self.p3<0 or self.p3>1:
                        return 'p3'
                if self.pdup<0:
                        return 'pdup'


def xp(old, new, operation):
        """Print the individual operations that transform
        one name into a new one.
        """
        print old, '--(%s)-->'%operation, new


class Name:
        __doc__ = """This class represents a single name,
                        and for convenience, it also stores
                        the parameters that control
                        the processes that transform names."""

        def __init__(self, syllablelist, nprm):
                """Create a name from a list of its syllables (syllablelist)
                        and the adjustable parameters (nprm)."""
                self.sl = syllablelist
                self.np = nprm

        def __str__(self):
                """Represent the name as a string."""
                return '-'.join(self.sl)

        __repr__ = __str__


        def p1(self):
                """Process 1: Prepend a nickname."""
                o = Name(random.choice(Nicknames) + self.sl, self.np)
                # xp(self, o, 'prepend')
                return o

        def p2(self):
                """Process 2: Append a placename or occupation."""
                o = Name(self.sl + random.choice(PlaceOcc), self.np)
                # xp(self, o, 'append')
                return o


        def p3(self):
                """Process 3: Drop syllables."""
                ns = len(self.sl)
                if ns <= 2:
                        # If the name is already short, just return a copy.
                        return Name(self.sl, self.np)

                while 1:
                        # Try to delete a range of syllables, and see if
                        # it leaves at least two syllables.
                        dropstart = random.randint(0, ns-1)
                        dropend = random.randint(1, ns-1)
                        if dropstart <= dropend and dropstart + (ns-dropend) >= 2:
                                break    # Yes! An acceptable drop.
                o = Name(self.sl[:dropstart]+self.sl[dropend:], self.np)
                # xp(self, o, 'drop')
                return o
```

```
    def evolve(self, namedict):
        """This generates the next generation's form of the
        current name."""

        # print 'NN----------:', self.sl
        while 1:
            x = random.random()
            tmp = Name(self.sl, self.np)
            if x < self.np.p1:
                tmp = tmp.p1()
            if x < self.np.p2:
                tmp = tmp.p2()
            if x < self.np.p3 * len(self.sl):
                tmp = tmp.p3()

            # Check to see if the new name duplicates other names already
            # out in the population.  If so, how many?    Also, does
            # it duplicate a place or occupational name?
            dups = namedict.get(str(tmp), 0) + 100000*Placedict.get(str(tmp), 0)
            if random.random() > self.np.pdup * float(dups)/float(len(namedict)):
                break    # Good enough!
            # print '--------TOO COMMON'
        return tmp


    def __cmp__(self, other):
        """Compare two names."""
        return cmp(self.sl, other.sl)


def generation(namelist):
    """Computes the names in generation N+1 from the array that is passed
    into it (generation N)."""

    namedict = {}
    for t in namelist:
        namedict[str(t)] = namedict.get(str(t), 0) + 1
    N = len(namelist)
    nnl = []
    for i in range(N):
        # We randomly choose names to breed.
        # Some names will therefore have no descendents;
        # some will have more than one.
        parent = random.choice(namelist)
        nn = parent.evolve(namedict)
        nnl.append(nn)
    return nnl


def print_statistics(namelist):
    """A helper function to let you watch the evolution
    of the statistical distribution of names.   It prints
    out some summary statistics; run() call it every generation.
    """
```

```
        N = len(namelist)
        lenhist = {}
        smith = 0
        for name in namelist:
                ln = len(name.sl)
                lenhist[ln] = lenhist.get(ln, 0) + 1
                smith += 'smith' in name.sl
        for l in range(10):
                print '#LEN:', l, '%.3f' % (lenhist.get(l, 0)/float(N))
        print '#SMITH:', smith/float(N)


def write_histogram(namelist, nfd):
        """A helper function: it writes a histogram of names
        to a file, to allow debugging.
        """
        hist = {}
        for n in namelist:
                sn = str(n)
                hist[sn] = hist.get(sn, 0) + 1
        histlist = [ (v, k) for (k, v) in hist.items() ]
        histlist.sort()
        histlist.reverse()
        for (v, k) in histlist:
                if v > 1:
                        nfd.writelines('%s %d\n' % ( k, v) )


def run(N, prms=None):
        """This function runs and prints 20 generations of statistics."""
        np = ModelParameters(prms)

        # Set up the names in the first generation:
        namelist = []
        for i in range(N):
                namelist.append( Name(random.choice(Nicknames+PlaceOcc), np) )

        # Run the simulation:
        for t in range(20):
                print '# ----GENERATION------', t
                namelist = generation(namelist)
                print_statistics(namelist)

        # Print the most common names in the final generation:
        import sys
        write_histogram(namelist, sys.stdout)



def resid(x, N):
        """This function is used to find the best-fit values
        of the adjustable parameters.
        It is called by an external script (not supplied)
        that adjusts the parameters, calls resid(),
        and looks to see whether or not the new model
        (based on the adjusted parameters) is a better
        or worse fit to our data.

        Argument x is an array of parameters to control the name generation
        process.
        Argument N is the number of names to simulate.
        """

        print 'prms=', x
        np = ModelParameters(x)
        if np.not_ok():
                return None        # Give up, if parameters are silly.

        # Compute a list of names:
        namelist = []
        NNPO = Nicknames + PlaceOcc
        for i in range(N):
                namelist.append( Name(random.choice(NNPO), np) )
        for t in range(20):
                namelist = generation(namelist)

        # Compute statistics from the list of names:
        lenhist = {}
        smith = 0
        for n in namelist:
                ln = len(n.sl)
                lenhist[ln] = lenhist.get(ln, 0) + 1
                smith += 'smith' in n.sl

        # The data:
        datasmith = 0.006
        data = [None, 0, 0.036, 0.20, 0.46, 0.23, 0.05, 0.009, 0.002, 0.0002]

        # Compare the data to the statistics from the simulation:
        o = [ math.log((smith/float(N))/datasmith) ]
        for l in range(2,10):
                o.append(math.log( (lenhist.get(l, 0)/float(N)) / data[l]))

        write_histogram(namelist, open('names.txt', 'w'))
        print 'r=', o
        # Return an array of the 'differences' between the model
        # statistics and the data.
        return [10*r for r in o]



def start(arglist):
        """Sets the starting position for the search to find
        the best-fit adjustable parameters.   This is used
        when optimizing the parameters; it is called by an external
        script."""
        return Numeric.array([0.25, 0.25, 0.25, 2], Numeric.Float)
```

```
def V(start):
        """Sets the initial region over which to search for the
        the best-fit adjustable parameters.   This is used
        when optimizing the parameters; it is called by an external
        script."""
        return Numeric.array([[1, 0, 0, 0], [0, 0.1, 0, 0], [0, 0, 1, 0],
                              [0, 0, 0, 1]], Numeric.Float)


NI = 1000          # Used for finding best-fit adjustable parameters.
c = 10000          # Used for finding best-fit adjustable parameters.

# Next, we compute a few things that will speed up the computation.
# First, a dictionary of place names, to allow us to rapidly decide
# whether or not a newly generated name matches a place name:
Placedict = {}
for p in Place:
        Placedict[str(Name(p, None))] = 1

# Second, we need an array of place or occupational names:
PlaceOcc = Place + Occ

if __name__ == '__main__':
        # Begin the computation.   The first argument is the
        # size of the population; the second argument
        # (an array) are the values of the adjustable parameters
        # that we want to use.
        run(10000, [0.43172252,  0.00283817,  0.13898237,  0.28479306])
```