MASSIMO DI PIERRO

ANNOTATED ALGORITHMS IN PYTHON

WITH APPLICATIONS IN PHYSICS, BIOLOGY, AND FINANCE (2ND ED)

EXPERTS4SOLUTIONS

5
Probability and Statistics

5.1 Probability

Probability derives from the Latin *probare* (to prove or to test). The word probably means roughly "likely to occur" in the case of possible future occurrences or "likely to be true" in the case of inferences from evidence. See also probability theory.

What mathematicians call probability is the mathematical theory we use to describe and quantify uncertainty. In a larger context, the word *probability* is used with other concerns in mind. Uncertainty can be due to our ignorance, deliberate mixing or shuffling, or due to the essential randomness of Nature. In any case, we measure the uncertainty of events on a scale from zero (impossible events) to one (certain events or no uncertainty).

There are three standard ways to define probability:

- (frequentist) Given an experiment and a set of possible outcomes S, the probability of an event $A \subset S$ is computed by repeating the experiment N times, counting how many times the event A is realized, N_A, then taking the limit

$$\text{Prob}(A) \equiv \lim_{N \to \infty} \frac{N_A}{N} \tag{5.1}$$

This definition actually requires that one performs an experiment, if not an infinite, then a number of times.

- (a priori) Given an experiment and a set of possible outcomes S with cardinality $c(S)$, the probability of an event $A \subset S$ is defined as

$$\text{Prob}(A) \equiv \frac{c(A)}{c(S)} \tag{5.2}$$

  This definition is ambiguous because it assumes that each "atomic" event $x \in S$ has the same a priori probability and therefore the definition itself is circular. Nevertheless we use this definition in many practical circumstances. What is the probability that when rolling a dice we will get an even number? The space of possible outcomes is $S = \{1, 2, 3, 4, 5, 6\}$ and $A = \{2, 4, 6\}$ therefore $\text{Prob}(A) = c(A)/c(S) = 3/6 = 1/2$. This analysis works for an ideal die and ignores the fact that a real dice may be biased. The former definition takes into account this possibility, whereas the latter does not.

- (axiomatic definition) Given an experiment and a set of possible outcomes S, the probability of an event $A \subset S$ is a number $\text{Prob}(A) \in [0, 1]$ that satisfies the following conditions: $\text{Prob}(S) = 1$; $\text{Prob}(A_1 \cup A_2) = \text{Prob}(A_1) + \text{Prob}(A_2)$ if $A_1 \cap A_2 = 0$.

In some sense, probability theory is a physical theory because it applies to the physical world (this is a nontrivial fact). While the axiomatic definition provides the mathematical foundation, the a priori definition provides a method to make predictions based on combinatorics. Finally the *frequentist* definition provides an experimental technique to confront our predictions with experiment (is our dice a perfect dice, or is it biased?).

We will differentiate between an "atomic" event defined as an event that can be realized by a single possible outcome of our experiment and a general event defined as a subset of the space of all possible outcomes. In the case of a dice, each possible number (from 1 to 6) is an event and is also an atomic event. The event of getting an even number is an event but not an atomic event because it can be realized in three possible ways.

The axiomatic definition makes it easy to prove theorems, for example,

If $S = A \cup A^c$ *and* $A \cap A^c = 0$ *then* $Prob(A) = 1 - Prob(A^c)$

Python has a module called `random` that can generate random numbers, and we can use it to perform some experiments. Let's simulate a dice with six possible outcomes. We can use the frequentist definition:

Listing 5.1: in file: `nlib.py`

```
>>> import random
>>> S = [1,2,3,4,5,6]
>>> def Prob(A, S, N=1000):
...     return float(sum(random.choice(S) in A for i in xrange(N)))/N
>>> Prob([6],S)
0.166
>>> Prob([1,2],S)
0.308
```

Here `Prob(A)` computes the probability that the event is set `A` using `N=1000` simulated experiments. The `random.choice` function picks one of the choices at random with equal probability.

We can compute the same quantity using the a priori definition:

Listing 5.2: in file: `nlib.py`

```
>>> def Prob(A, S): return float(len(A))/len(S)
>>> Prob([6],S)
0.16666666666666666
>>> Prob([1,2],S)
0.3333333333333333
```

As stated before, the latter is more precise because it produces results for an "ideal" dice while the frequentist's approach produces results for a real dice (in our case, a simulated dice).

5.1.1 Conditional probability and independence

We define $\text{Prob}(A|B)$ as the probability of event A given event B, and we write

$$\text{Prob}(A|B) \equiv \frac{\text{Prob}(AB)}{\text{Prob}(B)} \tag{5.3}$$

where $\text{Prob}(AB)$ is the probability that A and B both occur and $\text{Prob}(B)$ is the probability that B occurs. Note that if $\text{Prob}(A|B) = \text{Prob}(A)$, then we say that A and B are independent. From eq.(5.3) we conclude $\text{Prob}(AB) =$

Prob(A)Prob(B); therefore the probability that two independent events occur is the product of the probability that each individual event occurs.

We can experiment with conditional probability using Python. Let's consider two dices, X and Y. The space of all possible outcomes is given by $S^2 = S \times S$. And we are interested in the probability of the second die giving a 6 given that the first dice is also a 6:

Listing 5.3: in file: `nlib.py`

```
>>> def cross(u,v): return [(i,j) for i in u for j in v]
>>> def Prob_conditional(A, B, S): return Prob(cross(A,B),cross(S,S))/Prob(B,S)
>>> Prob_conditional([6],[6],S)
0.16666666666666666
```

Because we are only considering cases in which the second die is 6, we will pretend that when the second die is 1 through 5 didn't occur. Not surprisingly, we find that `Prob_conditional([6],[6],S)` produces the same result as `Prob([6],S)` because the two dices are independent.

In fact, we say that two sets of events A and B are independent if and only if $\mathrm{P}(A|B) = \mathrm{P}(A)$.

5.1.2 Discrete random variables

If S is in the space of all possible outcomes of an experiment and we associate an integer number X to each element of S, we say that X is a *discrete random variable*. If X is a discrete variable, we define $p(x)$, the *probability mass function* or *distribution*, as the probability that $X = x$:

$$p(x) \equiv \mathrm{Prob}(X = x) \tag{5.4}$$

We also define the *expectation value* of any function of a discrete random variable $f(X)$ as

$$E[f(X)] \equiv \sum_i f(x_i)p(x_i) \tag{5.5}$$

where i loops all possible variables x_i of the random variable X.

For example, if X is the random variable associated with the outcome of

rolling a dice, $p(x) = 1/6$ if $x = 1, 2, 3, 4, 5$ or 6 and $p(x) = 0$ otherwise:

$$E[X] = \sum_i x_i p(x_i) = \sum_{x_i \in \{1,2,3,4,5,6\}} x_i \frac{1}{6} = 3.5 \tag{5.6}$$

and

$$E[(X-3.5)^2] = \sum_i (x_i - 3.5)^2 p(x_i) = \sum_{x_i \in \{1,2,3,4,5,6\}} (x_i - 3.5)^2 \frac{1}{6} = 2.9167 \tag{5.7}$$

We call $E[X]$ the *mean* of X and usually denote it with μ_X. We call $E[(X - \mu_X)^2]$ the *variance* of X and denote it with σ_X^2. Note that

$$\sigma_X = E[X^2] - E[X]^2 \tag{5.8}$$

For discrete random variables, we can implement these definitions as follows:

Listing 5.4: in file: `nlib.py`

```
def E(f,S): return float(sum(f(x) for x in S))/(len(S) or 1)
def mean(X): return E(lambda x:x, X)
def variance(X): return E(lambda x:x**2, X) - E(lambda x:x, X)**2
def sd(X): return sqrt(variance(X))
```

which we can test with a simulated experiment:

Listing 5.5: in file: `nlib.py`

```
>>> S = [random.random()+random.random() for i in xrange(100)]
>>> print mean(S)
1.000...
>>> print sd(S)
0.4...
```

As another example, let's consider a simple bet on a dice. We roll the dice once and win $20 if the dice returns 6; we lose $5 otherwise:

Listing 5.6: in file: `nlib.py`

```
>>> S = [1,2,3,4,5,6]
>>> def payoff(x): return 20.0 if x==6 else -5.0
>>> print E(payoff,S)
-0.83333...
```

The average expected payoff is $-0.83\ldots$, which means that on average, we should expect to lose 83 cents at this game.

5.1.3 Continuous random variables

If S is the space of all possible outcomes of an experiment and we associate a real number X with each element of S, we say that X is a *continuous random variable*. We also define a *cumulative distribution function* $F(x)$ as the probability that $X \leq x$:

$$F(x) \equiv \text{Prob}(X \leq x) \tag{5.9}$$

If S is a continuous set and X is a continuous random variable, then we define a *probability density* or *distribution* $p(x)$ as

$$p(x) \equiv \frac{dF(x)}{dx} \tag{5.10}$$

and the probability that X falls into an interval $[a, b]$ can be computed as

$$\text{Prob}(a \leq X \leq b) = \int_a^b p(x)dx \tag{5.11}$$

We also define the *expectation value* of any function of a random variable $f(X)$ as

$$E[f(X)] = \int_{-\infty}^{\infty} f(x)p(x)dx \tag{5.12}$$

For example, if X is a uniform random variable (probability density $p(x)$ equal to 1 if $x \in [0, 1]$, equal to 0 otherwise)

$$E[X] = \int_{-\infty}^{\infty} xp(x)dx = \int_0^1 xdx = \frac{1}{2} \tag{5.13}$$

and

$$E[(X - \frac{1}{2})^2] = \int_{-\infty}^{\infty} (x - \frac{1}{2})^2 p(x)dx = \int_0^1 (x^2 - x + \frac{1}{4})dx = \frac{1}{12} \tag{5.14}$$

We call $E[X]$ the **mean** of X and usually denote it with μ_X. We call $E[(X - \mu_X)^2]$ the **variance** of X and denote it with σ_X^2. Note that

$$\sigma_X^2 = E[X^2] - E[X]^2 \tag{5.15}$$

By definition,

$$F(\infty) \equiv \text{Prob}(X \leq \infty) = 1 \tag{5.16}$$

therefore

$$\text{Prob}(-\infty \leq X \leq \infty) = \int_{-\infty}^{\infty} p(x)dx = 1 \tag{5.17}$$

The distribution p is always normalized to 1.

Moreover,

$$E[aX + b] = \int_{-\infty}^{\infty} (ax + b)p(x)dx \tag{5.18}$$

$$= a\int_{-\infty}^{\infty} xp(x)dx + b\int_{-\infty}^{\infty} p(x)dx \tag{5.19}$$

$$= aE[X] + b \tag{5.20}$$

therefore $E[X]$ is a linear operator.

One important consequence of all these formulas is that if we have a function f and a domain $[a, b]$, we can compute its integral by choosing p to be a uniform distribution with values exclusively between a and b:

$$E[f] = \int_{-\infty}^{\infty} f(x)p(x)dx = \frac{1}{b-a}\int_{a}^{b} f(x)dx \tag{5.21}$$

We can also compute the same integral by using the definition of expectation value for a discrete distribution:

$$E[f] = \sum_{x_i} f(x_i)p(x_i) = \frac{1}{N}\sum_{x_i} f(x_i) \tag{5.22}$$

where x_i are N random points drawn from the uniform distribution p defined earlier. In fact, in the large N limit,

$$\lim_{N\to\infty} \frac{1}{N}\sum_{x_i} f(x_i) = \int_{-\infty}^{\infty} f(x)p(x)dx = \frac{1}{b-a}\int_{a}^{b} f(x)dx \tag{5.23}$$

We can verify the preceding relation numerically for a special case:

Listing 5.7: in file: nlib.py

```
>>> from math import sin, pi
>>> def integrate_mc(f,a,b,N=1000):
...     return sum(f(random.uniform(a,b)) for i in xrange(N))/N*(b-a)
>>> print integrate_mc(sin,0,pi,N=10000)
2.000....
```

This is the simplest case of Monte Carlo integration, which is the subject of a following chapter.

5.1.4 Covariance and correlations

Given two random variables, X and Y, we define the covariance (cov) and the correlation (corr) between them as

$$\text{cov}(X,Y) \equiv E[(X-\mu_X)(Y-\mu_Y)] = E[XY] - E[X]E[Y] \tag{5.24}$$

$$\text{corr}(X,Y) \equiv \text{cov}(X,Y)/(\sigma_X\sigma_Y) \tag{5.25}$$

Applying the definitions:

$$E[XY] = \int\int xyp(x,y)dxdy \tag{5.26}$$

$$= \int\int xyp(x)p(y)dxdy \tag{5.27}$$

$$= \left[\int xp(x)dx\right]\left[\int yp(y)dy\right] \tag{5.28}$$

$$= E[X]E[Y] \tag{5.29}$$

therefore

$$\text{cov}(X,Y) = E[XY] - E[X]E[Y] = 0. \tag{5.30}$$

Therefore

$$\sigma^2_{X+Y} = \sigma^2_X + \sigma^2_Y + 2\text{cov}(X,Y) \tag{5.31}$$

and if X and Y are independent, then $\text{cov}(X,Y) = \text{corr}(X,Y) = 0$.

Notice that the reverse is not true. Even if the correlation and the covariance are zero, X and Y may be dependent.

Moreover,

$$\begin{aligned}\operatorname{cov}(X,Y) &= E[(X-\mu_X)(Y-\mu_Y)] && (5.32)\\ &= E[(X-\mu_X)(\pm X \mp \mu_X)] && (5.33)\\ &= \pm E[(X-\mu_X)(X-\mu_X)] && (5.34)\\ &= \pm\sigma_X^2 && (5.35)\end{aligned}$$

Therefore, if X and Y are completely correlated or anti-correlated ($Y = \pm X$), then $\operatorname{cov}(X,Y) = \pm\sigma_X^2$ and $\operatorname{corr}(X,Y) = \pm 1$. Notice that the correlation lies always in the range $[-1,1]$.

Finally, notice that for uncorrelated random variables X_i,

$$E[\sum_i a_i X_i] = \sum_i a_i E[X_i] \quad (5.36)$$

$$E[(\sum_i X_i)^2] = \sum_i E[X_i^2] \quad (5.37)$$

We can define covariance and correlation for discrete distributions:

Listing 5.8: in file: `nlib.py`

```
def covariance(X,Y):
    return sum(X[i]*Y[i] for i in xrange(len(X)))/len(X) - mean(X)*mean(Y)
def correlation(X,Y):
    return covariance(X,Y)/sd(X)/sd(Y)
```

Here is an example:

Listing 5.9: in file: `nlib.py`

```
>>> X = []
>>> Y = []
>>> for i in xrange(1000):
...     u = random.random()
...     X.append(u+random.random())
...     Y.append(u+random.random())
>>> print mean(X)
0.989780352018
>>> print sd(X)
0.413861115381
>>> print mean(Y)
1.00551523013
>>> print sd(Y)
0.404909628555
```

```
>>> print covariance(X,Y)
0.0802804358268
>>> print correlation(X,Y)
0.479067813484
```

5.1.5 Strong law of large numbers

If $X_1, X_2, \ldots, X_n$ are a sequence of independent and identically distributed random variables with $E[X_i] = \mu$ and finite variance, then

$$\lim_{n\to\infty} \frac{X_1 + X_2 + \cdots + X_n}{n} = \mu \tag{5.38}$$

This theorem means that "the average of the results obtained from a large number of trials should be close to the expected value, and will tend to become closer as more trials are performed." The name of this law is due to Poisson [43].

5.1.6 Central limit theorem

This is one of the most important theorems concerning distributions [44]: if $X_1, X_2, \ldots, X_n$ are a sequence of random variables with finite means, μ_i, and finite variance, σ_i^2, then

$$Y = \lim_{N\to\infty} \frac{1}{N} \sum_{i=0}^{i<N} X_i \tag{5.39}$$

follows a Gaussian distribution with mean and variance:

$$\mu = \lim_{N\to\infty} \frac{1}{N} \sum_{i=0}^{i<N} \mu_i \tag{5.40}$$

$$\sigma^2 = \lim_{N\to\infty} \frac{1}{N} \sum_{i=0}^{i<N} \sigma_i^2 \tag{5.41}$$

We can numerically verify this for the simple case in which X_i are uniform random variables with mean equal to 0:

Listing 5.10: in file: `nlib.py`

```
>>> def added_uniform(n): return sum([random.uniform(-1,1) for i in xrange(n)])/
    n
>>> def make_set(n,m=10000): return [added_uniform(n) for j in xrange(m)]
>>> Canvas(title='Central Limit Theorem',xlab='y',ylab='p(y)'
...       ).hist(make_set(1),legend='N=1').save('images/central1.png')
>>> Canvas(title='Central Limit Theorem',xlab='y',ylab='p(y)'
...       ).hist(make_set(2),legend='N=2').save('images/central3.png')
>>> Canvas(title='Central Limit Theorem',xlab='y',ylab='p(y)'
...       ).hist(make_set(4),legend='N=4').save('images/central4.png')
>>> Canvas(title='Central Limit Theorem',xlab='y',ylab='p(y)'
...       ).hist(make_set(8),legend='N=8').save('images/central8.png')
```

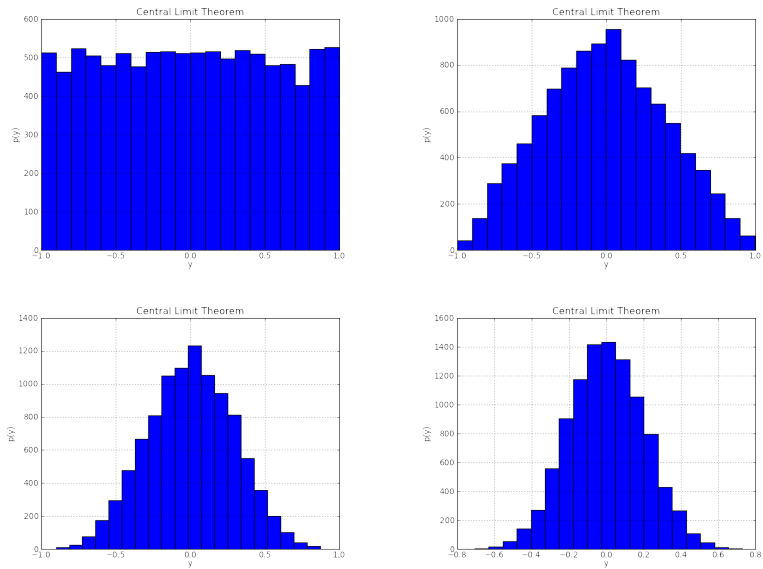

Figure 5.1: Example of distributions for sums of 1, 2, 4, and 8 uniform random variables. The more random variables are added, the better the result approximates a Gaussian distribution.

This theorem is of fundamental importance for stochastic calculus. Notice that the theorem does not apply when the X_i follow distributions that do not have a finite mean or a finite variance.

Distributions that do not follow the central limit are called *Levy distribu-*

tions. They are characterized by fat tails for the form

$$p(x) \underset{x\to\infty}{\sim} \frac{1}{|x|^{1+\alpha}}, \qquad 0 < \alpha < 2 \tag{5.42}$$

An example if the Pareto distribution.

5.1.7 Error in the mean

One consequence of the Central Limit Theorem is a useful formula for evaluating the error in the mean. Let's consider the case of N repeated experiments with outcomes X_i. Let's also assume that each X_i is supposed to be equal to an unknown value μ, but in practice, $X_i = \mu + \varepsilon$, where ε is a random variable with Gaussian distribution centered at zero. One could estimate μ by $\mu = E[X] = \frac{1}{N}\Sigma_i X_i$. In this case, statistical error in the mean is given by

$$\delta\mu = \sqrt{\frac{\sigma^2}{N}} \tag{5.43}$$

where $\sigma^2 = E[(X-\mu)^2] = \frac{1}{N}\Sigma_i (X_i - \mu)^2$.

5.2 Combinatorics and discrete random variables

Often, to compute the probability of discrete random variables, one has to confront the problem of calculating the number of possible finite outcomes of an experiment. Often this problem is solved by combinatorics.

5.2.1 Different plugs in different sockets

If we have n different plugs and m different sockets; in how many ways can we place the plugs in the sockets?

- Case 1, $n \geq m$. All sockets will be filled. We consider the first socket, and we can select any of the n plugs (n combinations). We consider the second socket, and we can select any of the remaining $n-1$ plugs

($n-1$ combinations), and so on, until we are left with no free sockets and $n-m$ unused plugs; therefore there are

$$n!/(n-m)! = n(n-1)(n-2)\dots(n-m+1) \tag{5.44}$$

combinations.

- Case 2, $n \leq m$. All plugs have to be used. We consider the first plug, and we can select any of the m sockets (m combinations). We consider the second plug, and we can select any of the remaining $m-1$ sockets ($m-1$ combinations), and so on, until we are left with no spare plugs and $m-n$ free sockets; therefore there are

$$m!/(m-n)! = m(m-1)(m-2)\dots(m-n+1) \tag{5.45}$$

combinations. Note that if $m = n$ then case 1 and case 2 agree, as expected.

5.2.2 Equivalent plugs in different sockets

If we have n equivalent plugs and m different sockets, in how many ways can we place the plugs in the sockets?

- Case 1, $n \geq m$. All sockets will be filled. We cannot distinguish one combination from the other because all plugs are the same. There is only one combination.
- Case 2, $n \leq m$. All plugs have to be used but not all sockets. There are $m!/(m-n)!$ ways to fill the sockets with different plugs, and there are $n!$ ways to arrange the plugs within the same filled sockets. Therefore there are

$$\binom{m}{n} = \frac{m!}{(m-n)!n!} \tag{5.46}$$

ways to place n equivalent plugs into m different sockets. Note that if $m = n$

$$\binom{n}{n} = \frac{n!}{(n-n)!n!} = 1 \tag{5.47}$$

in agreement with case 1.

Here is another example. A club has 20 members and has to elect a president, a vice president, a secretary, and a treasurer. In how many different ways can they select the four officeholders? Think of each office as a socket and each person as a plug; therefore the number combination is $20!/(20-4)! \simeq 1.2 \times 10^5$.

5.2.3 Colored cards

We have 52 cards, 26 black and 26 red. We shuffle the cards and pick three.

- What is the probability that they are all red?

$$\text{Prob}(3red) = \frac{26}{52} \times \frac{25}{51} \times \frac{24}{50} = \frac{2}{17} \tag{5.48}$$

- What is the probability that they are all black?

$$\text{Prob}(3black) = \text{Prob}(3red) = \frac{2}{17} \tag{5.49}$$

- What is the probability that they are not all black or all red?

$$\begin{aligned} \text{Prob}(mixture) &= 1 - \text{Prob}(3red \cup 3black) && (5.50)\\ &= 1 - \text{Prob}(3red) - \text{Prob}(3black) && (5.51)\\ &= 1 - 2\frac{2}{17} && (5.52)\\ &= \frac{13}{17} && (5.53) \end{aligned}$$

Here is an example of how we can simulate the deck of cards using Python to compute an answer to the last questions:

Listing 5.11: in file: `tests.py`

```
>>> def make_deck(): return [color for i in xrange(26) for color in ('red','
    black')]
>>> def make_shuffled_deck(): return random.shuffle(make_deck())
>>> def pick_three_cards(): return make_shuffled_deck()[:3]
>>> def simulate_cards(n=1000):
```

```
...     counter = 0
...     for k in xrange(n):
...         c = pick_three_cards()
...         if not (c[0]==c[1] and c[1]==c[2]): counter += 1
...     return float(counter)/n
>>> print simulate_cards()
```

5.2.4 Gambler's fallacy

The typical error in computing probabilities is mixing a priori probability with information about past events. This error is called the *gambler's fallacy* [45]. For example, we consider the preceding problem. We see the first two cards, and they are both red. What is the probability that the third one is also red?

- **Wrong answer**: The probability that they are all red is $\text{Prob}(3red) = 2/17$; therefore the probability that the third one is also $2/17$.
- **Correct answer**: Because we know that the first two cards are red, the third card must belong to a set of (26 black cards + 24 red cards); therefore the probability that it is red is

$$\text{Prob}(red) = \frac{24}{24+26} = \frac{12}{25} \tag{5.54}$$

6

Random Numbers and Distributions

In the previous chapters, we have seen how using the Python `random` module, we can generate uniform random numbers. This module can also generate random numbers following other distributions. The point of this chapter is to understand how random numbers are generated.

6.1 Randomness, determinism, chaos and order

Before we proceed further, there are four important concepts that should be defined because of their implications:

- **Randomness** is the characteristic of a process whose outcome is unpredictable (e.g., at the moment I am writing this sentence, I cannot predict the exact time and date when you will be reading it).
- **Determinism** is the characteristic of a process whose outcome can be predicted from the initial conditions of the system (e.g., if I throw a ball from a known position, at a known velocity and in a known direction, I can predict—calculate—its entire future trajectory).
- **Chaos** is the emergence of randomness from order [46] (e.g., if I am on the top of a hill and I throw the ball in a vertical direction, I cannot predict on which side of the hill it is going to end up). Even if the equations that describe a phenomenon are known and are determinis-

tic, it may happen that a small variation in the initial conditions causes a large difference in the possible deterministic evolution of the system. Therefore the outcome of a process may depend on a tiny variation of the initial parameters. These variations may not be measurable in practice, thus making the process unpredictable and chaotic. Chaos is generally regarded as a characteristic of some differential equations.

- **Order** is the opposite of chaos. It is the emergence of regular and reproducible patterns from a process that, in itself, may be random or chaotic (e.g., if I keep throwing my ball in a vertical direction from the top of a hill and I record the final location of the ball, I eventually find a regular pattern, a probability distribution associated with my experiment, which depends on the direction of the wind, the shape of the hill, my bias in throwing the ball, etc.).

These four concepts are closely related, and they do not necessarily come in opposite pairs as one would expect.

A deterministic process may cause chaos. We can use chaos to generate randomness (we will see examples when covering random number generation). We can study randomness and extract its ordered properties (probability distributions), and we can use randomness to solve deterministic problems (Monte Carlo) such as computing integrals and simulating a system.

6.2 Real randomness

Note that randomness does not necessarily come from chaos. Randomness exists in nature [47][48]. For example, a radioactive atom "decays" into a different atom at some random point in time. For example, an atom of carbon 14 decays into nitrogen 14 by emitting an electron and a neutrino

$$ {}^{14}_{6}C \longrightarrow {}^{14}_{7}N + e^{-} + \overline{\nu}_e \tag{6.1} $$

at some random time t; t is unpredictable. It can be proven that the randomness in the nuclear decay time is not due to any underlying deterministic process. In fact, constituents of matter are described by quantum

physics, and randomness is a fundamental characteristic of quantum systems. Randomness is not a consequence of our ignorance.

This is not usually the case for macroscopic systems. Typically the randomness we observe in some macroscopic systems is not always a consequence of microscopic randomness. Rather, order and determinism emerge from the microscopic randomness, while chaos originates from the complexity of the system.

Because randomness exists in nature, we can use it to produce random numbers with any desired distribution. In particular, we want to use the randomness in the decay time of radioactive atoms to produce random numbers with uniform distribution. We assemble a system consisting of many atoms, and we record the time when we observe atoms decay:

$$t_0, t_1, t_2, t_3, t_4, t_5 \ldots, \tag{6.2}$$

One could study the probability distribution of the t_i and find that it follows an exponential probability distribution like

$$\text{Prob}(t_i = t) = \lambda e^{-\lambda t} \tag{6.3}$$

where $t_0 = 1/\lambda$ is the decay time characteristic of the particular type of atom. One characteristic of this distribution is that it is a memoryless process: t_i does not depend on t_{i-1} and therefore the probability that $t_i > t_{i-1}$ is the same as the probability that $t_i < t_{i-1}$.

6.2.1 Memoryless to Bernoulli distribution

Given the sequence $\{t_i\}$ with exponential distribution, we can build a random sequence of zeros and ones (Bernoulli distribution) by applying the following formula, known as the Von Neumann procedure [49]:

$$x_i = \begin{cases} 1 & \text{if } t_i > t_{i-1} \\ 0 & \text{otherwise} \end{cases} \tag{6.4}$$

Note that the procedure can be applied to map any random sequence into a Bernoulli sequence even if the numbers in the original sequence do not follow an exponential distribution, as long as t_i is independent of t_j for any $j < i$ (memoryless distribution).

6.2.2 Bernoulli to uniform distribution

To map a Bernoulli distribution into a uniform distribution, we need to determine the precision (resolution in number of bits) of the numbers we wish to generate. In this example, we will assume 8 bits.

We can think of each number as a point in a [0,1) segment. We generate the uniform number by making a number of choices: we break the segment in two and, according to the value of the binary digit (0 or 1), we select the first part or the second part and repeat the process on the subsegment. Because at each stage we break the segment into two parts of equal length and we select one or the other with the same probability, the final distribution of the selected point is uniform. As an example, we consider the Bernoulli sequence

$$01011110110101010111011010 \tag{6.5}$$

and we perform the following steps:

- break the sequence into chunks of 8 bits

$$01011110\text{-}11010101\text{-}01110110\text{-}..... \tag{6.6}$$

- map each chunk $a_0a_1a_2a_3a_4a_5a_6a_7$ into $x = \sum_{k=0}^{k<8} a_k/2^{k+1}$ thus obtaining:

$$0.3671875 - 0.83203125 - 0.4609375 - \ldots \tag{6.7}$$

A uniform random number generator is usually the first step toward building any other random number generator.

Other physical processes can be used to generate real random numbers using a similar process. Some microprocessors can generate random num-

bers from random temperature fluctuations. An unpredictable source of randomness is called an *entropy source*.

6.3 Entropy generators

The Linux/Unix operating system provides its own entropy source accessible via "/dev/urandom." This data source is available in Python via `os.urandom()`.

Here we define a class that can access this entropy source and use it to generate uniform random numbers. It follows the same process outlined for the radioactive days:

```
class URANDOM(object):
    def __init__(self, data=None):
        if data: open('/dev/urandom','wb').write(str(data))
    def random(self):
        import os
        n = 16
        random_bytes =  os.urandom(n)
        random_integer = sum(ord(random_bytes[k])*256**k for k in xrange(n))
        random_float = float(random_integer)/256**n
```

Notice how the constructor allows us to further randomize the data by contributing input to the entropy source. Also notice how the `random()` method reads 16 bites from the stream (using `os.urandom()`), converts each into 8-bit integers, combines them into a 128-bit integer, and then converts it to a float by dividing by 256^{16}.

6.4 Pseudo-randomness

In many cases we do not have a physical device to generate random numbers, and we require a software solution. Software is deterministic, the outcome is reproducible, therefore it cannot be used to generate randomness, but it can generate pseudo-randomness. The outputs of pseudo random number generators are not random, yet they may be considered random for practical purposes. John von Neumann observed in 1951 that "anyone who considers arithmetical methods of producing random digits

is, of course, in a state of sin." (For attempts to generate "truly random" numbers, see the article on hardware random number generators.) Nevertheless, pseudo random numbers are a critical part of modern computing, from cryptography to the Monte Carlo method for simulating physical systems.

Pseudo random numbers are relatively easy to generate with software, and they provide a practical alternative to random numbers. For some applications, this is adequate.

6.4.1 Linear congruential generator

Here is probably the simplest possible pseudo random number generator:

$$x_i = (ax_{i-1} + c) \mathrm{mod} m \tag{6.8}$$

$$y_i = x_i / m \tag{6.9}$$

With the choice $a = 65539$, $c = 0$, and $m = 2^{31}$, this generator is called RANDU. It is of historical importance because it is implemented in the C `rand()` function. The RANDU generator is particularly fast because the modulus can be implemented using the finite 32-bit precision.

Here is a possible implementation for $c = 0$:

Listing 6.1: in file: `nlib.py`

```
class MCG(object):
    def __init__(self,seed,a=66539,m=2**31):
        self.x = seed
        self.a, self.m = a, m
    def next(self):
        self.x = (self.a*self.x) % self.m
        return self.x
    def random(self):
        return float(self.next())/self.m
```

which we can test with

```
>>> randu = MCG(seed=1071914055)
>>> for i in xrange(10): print randu.random()
...
```

The output numbers "look" random but are not truly random. Running the same code with the same seed generates the same output. Notice the following:

- PRNGs are typically implemented as a recursive expression that, given x_{i-1}, produces x_i.
- PRNGs have to start from an initial value, x_0, called the *seed*. A typical choice is to set the seed equal to the number of seconds from the conventional date and time "Thu Jan 01 01:00:00 1970." This is not always a good choice.
- PRNGs are periodic. They generate numbers in a finite set and then they repeat themselves. It is desirable to have this set as large as possible.
- PRNGs depend on some parameters (e.g., a and m). Some parameter choices lead to trivial random number generators. In general, some choices are better than others, and a few are optimal. In particular, the values of a and m determine the period of the random number generator. An optimal choice is the one with the longest period.

For a linear congruential generator, because of the `mod` operation, the period is always less than or equal to m. When c is nonzero, the period is equal to m only if c and m are relatively prime, $a-1$ is divisible by all prime factors of m, and $a-1$ is a multiple of 4 when m is a multiple of 4.

In the case of RANDU, the period is $m/4$. A better choice is using $a = 7^5$ and $m = 2^{31} - 1$ (known as the Mersenne prime number) because it can be proven that m is in fact the period of the generator:

$$x_i = (7^5 x_{i-1}) \text{mod} (2^{31} - 1) \tag{6.10}$$

Here are some examples of MCG used by various systems:

Source	m	a	c
Numerical Recipes	2^{32}	1664525	1013904223
glibc (used by GCC)	2^{32}	1103515245	12345
Apple CarbonLib	$2^{31}-1$	16807	0
java.util.Random	2^{48}	25214903917	11

When c is set to zero, a linear congruential generator is also called a multiplicative congruential generator.

6.4.2 Defects of PRNGs

The non-randomness of pseudo random number generators manifests itself in at least two different ways:

- The sequence of generated numbers is periodic, therefore only a finite set of numbers can come out of the generator, and many of the numbers will never be generated. This is not a major problem if the period is much larger (some order of magnitude) than the number of random numbers needed in the Monte Carlo computation.
- The sequence of generated numbers presents bias in the form of "patterns." Sometimes these patterns are evident, sometimes they are not evident. Patterns exist because the pseudo random numbers are not random but are generated using a recursive formula. The existence of these patterns may introduce a bias in Monte Carlo computations that use the generator. This is a nasty problem, and the implications depend on the specific case.

An example of pattern/bias is discussed in ref. [51] and can be seen in fig. 6.4.2.

6.4.3 Multiplicative recursive generator

Another modification of the multiplicative congruential generator is the following:

$$x_i = (a_1 x_{i-1} + a_2 x_{i-2} + \cdots + a_k x_{i-k}) \mathrm{mod} m \tag{6.11}$$

The advantage of this generator is that if m is prime, the period of this type of generator can be as big as $m^k - 1$. This is much larger than a simple multiplicative congruential generator.

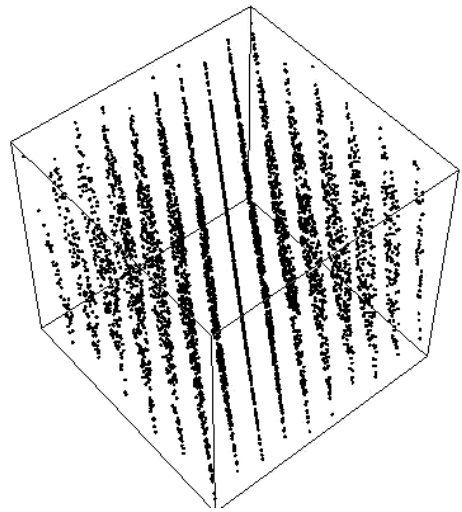

Figure 6.1: In this plot, each three consecutive random numbers (from RANDU) are interpreted as (x, y, z) coordinates of a random point. The image clearly shows the points are not distributed at random. Image from ref. [51].

An example is $a_1 = 107374182$, $a_2 = a_3 = a_4 = 0$, $a_5 = 104480$, and $m = 2^{31} - 1$, where the period is

$$(2^{31} - 1)^5 - 1 \simeq 4.56 \times 10^{46} \tag{6.12}$$

6.4.4 Lagged Fibonacci generator

$$x_i = (x_{i-j} + x_{i-k}) \mathrm{mod} m \tag{6.13}$$

This is similar to the multiplicative recursive generator earlier. If m is prime and $j \neq k$, the period can be as large as $m^k - 1$.

6.4.5 Marsaglia's add-with-carry generator

$$x_i = (x_{i-j} + x_{i-k} + c_i) \mathrm{mod} m \tag{6.14}$$

where $c_1 = 0$ and $c_i = 1$ if $(x_{i-1-j} + x_{i-1-k} + c_{i-1}) < m$, 0

6.4.6 Marsaglia's subtract-and-borrow generator

$$x_i = (x_{i-j} - x_{i-k} - c_i) \mathrm{mod} m \tag{6.15}$$

where $k > j > 0$, $c_1 = 0$, and $c_i = 1$ if $(x_{i-1-j} - x_{i-1-k} - c_{i-1}) < 0$, 0 otherwise.

6.4.7 Lüscher's generator

The Marsaglia's subtract-and-borrow is a very popular generator, but it is known to have some problems. For example, if we construct vector

$$v_i = (x_i, x_{i+1}, \ldots, x_{i+k}) \tag{6.16}$$

and the coordinates of the point v_i are numbers closer to each other then the coordinates of the point v_{i+k} are also close to each other. This indicates that there is an unwanted correlation between the points $x_i, x_{i+1}, \ldots, x_{i+k}$. Lüscher observed [50] that the Marsaglia's subtract-and-borrow is equivalent to a chaotic discrete dynamic system, and the preceding correlation dies off for points that distance themselves more than k. Therefore he proposed to modify the generator as follows: instead of taking all x_i numbers, read k successive elements of the sequence, discard $p - k$ numbers, read k numbers, and so on. The number p has to be chosen to be larger than k. When $p = k$, the original Marsaglia generator is recovered.

6.4.8 Knuth's polynomial congruential generator

$$x_i = (ax_{i-1}^2 + bx_{i-1} + c) \mathrm{mod} m \tag{6.17}$$

This generator takes the form of a more complex function. It makes it harder to guess one number in the sequence from the following numbers; therefore it finds applications in cryptography.

Another example is the Blum, Blum, and Shub generator:

$$x_i = x_{i-1}^2 \mathrm{mod} m \tag{6.18}$$

6.4.9 PRNGs in cryptography

Random numbers find many applications in cryptography. For example, consider the problem of generating a random password, a digital signature, or random encryption keys for the Diffie–Hellman and the RSA encryption schemes.

A cryptographically secure pseudo random number generator (CSPRNG) is a pseudo random number generator (PRNG) with properties that make it suitable for use in cryptography.

In addition to the normal requirements for a PRNG (that its output should pass all statistical tests for randomness) a CSPRNG must have two additional properties:

- It should be difficult to predict the output of the CSPRNG, wholly or partially, from examining previous outputs.
- It should be difficult to extract all or part of the internal state of the CSPRNG from examining its output.

Most PRNGs are not suitable for use as CSPRNGs. They must appear random in statistical tests, but they are not designed to resist determined mathematical reverse engineering.

CSPRNGs are designed explicitly to resist reverse engineering. There are a number of examples of CSPRNGs. Blum, Blum, and Shub has the strongest security proofs, though it is slow.

Many pseudo random number generators have the form

$$x_i = f(x_{i-1}, x_{i-2}, \ldots, x_{i-k}) \tag{6.19}$$

For example, the next random number depends on the past k numbers. Requirements for CSPRNGs used in cryptography are that

- Given $x_{i-1}, x_{i-2}, \ldots, x_{i-k}$, x_i can be computed in polynomial time, while
- Given $x_i, x_{i-2}, \ldots, x_{i-k}$, x_{i-1} must not be computable in polynomial time.

The first requirement means that the PRNG must be fast. The second requirement means that if a malicious agent discovers a random number used as a key, he or she cannot easily compute all previous keys generated using the same PRNG.

6.4.10 Inverse congruential generator

$$x_i = (ax_{i-1}^{-1} + c) \text{mod} m \tag{6.20}$$

where x_{i-1}^{-1} is the multiplicative inverse of x_{i-1} modulo m, for example, $x_{i-1}x_{i-1}^{-1} = 1 \text{mod} m$.

6.4.11 Mersenne twister

One of the best PRNG algorithms (because of its long period, uniform distribution, and speed) is the Mersenne twister, which produces a 53-bit random number, and it has a period of $2^{19937} - 1$ (this number is 6002 digits long!). The Python `random` module uses the Mersenne twister. Although discussing the inner working of this algorithm is beyond the scope of these notes, we provide a pure Python implementation of the Mersenne twister:

Listing 6.2: in file: `nlib.py`

```
class MersenneTwister(object):
    """
    based on:
    Knuth 1981, The Art of Computer Programming
    Vol. 2 (2nd Ed.), pp102]
    """
    def __init__(self,seed=4357):
        self.w = []   # the array for the state vector
        self.w.append(seed & 0xffffffff)
        for i in xrange(1, 625):
            self.w.append((69069 * self.w[i-1]) & 0xffffffff)
        self.wi = i
    def random(self):
        w = self.w
        wi = self.wi
        N, M, U, L = 624, 397, 0x80000000, 0x7fffffff
```

```
        K = [0x0, 0x9908b0df]
        y = 0
        if wi >= N:
            for kk in xrange((N-M) + 1):
                y = (w[kk]&U)|(w[kk+1]&L)
                w[kk] = w[kk+M] ^ (y >> 1) ^ K[y & 0x1]

            for kk in xrange(kk, N):
                y = (w[kk]&U)|(w[kk+1]&L)
                w[kk] = w[kk+(M-N)] ^ (y >> 1) ^ K[y & 0x1]
            y = (w[N-1]&U)|(w[0]&L)
            w[N-1] = w[M-1] ^ (y >> 1) ^ K[y & 0x1]
        wi = 0
        y = w[wi]
        wi += 1
        y ^= (y >> 11)
        y ^= (y << 7) & 0x9d2c5680
        y ^= (y << 15) & 0xefc60000
        y ^= (y >> 18)
        return (float(y)/0xffffffff )
```

In the above code, numbers starting with `0x` are represented in hexadecimal notation. The symbols `&`, `^`, `<<`, and `>>` are bitwise operators. `&` is a binary AND, `^` is a binary exclusive XOR, `<<` shifts all bits to the left and `>>` shifts all bits to the right. We refer to the Python official documentation for details.

6.5 Parallel generators and independent sequences

It is often necessary to generate many independent sequences.

For example, you may want to generate streams or random numbers in parallel using multiple machines and processes, and you need to ensure that the streams do not overlap.

A common mistake is to generate the sequences using the same generator with different seeds. This is not a safe procedure because it is not obvious if the seed used to generate one sequence belongs to the sequence generated by the other seed. The two sequences of random numbers are not independent, but they are merely shifted in respect to each other.

For example, here are two RANDU sequences generated with different

but dependent seeds:

seed	1071931562	50554362
y_0	0.252659081481	0.867315522395
y_1	0.0235412092879	0.992022250779
y_2	0.867315522395	0.146293803118
y_3	0.992022250779	0.949562561698
y_4	0.146293803118	0.380731142126
y_5	...	...

(6.21)

Note that the second sequence is the same as the first but shifted by two lines.

Three standard techniques for generating independent sequences are non-overlapping blocks, leapfrogging, and Lehmer trees.

6.5.1 Non-overlapping blocks

Let's consider one sequence of pseudo random numbers:

$$x_0, x_1, \ldots, x_k, x_{k+1}, \ldots, x_{2k}, x_{2k+1}, \ldots, x_{3k}, x_{3k+1}, \ldots, \tag{6.22}$$

One can break it into subsequences of k numbers:

$$x_0, x_1, \ldots, x_{k-1} \tag{6.23}$$

$$x_k, x_{k+1}, \ldots, x_{2k-1} \tag{6.24}$$

$$x_{2k}, x_{2k+1}, \ldots, x_{3k-1} \tag{6.25}$$

$$\ldots \tag{6.26}$$

If the original sequence is created with a multiplicative congruential generator

$$x_i = ax_{i-1} \bmod m \tag{6.27}$$

the subsequences can be generated independently because

$$x_{nk-1} = a^{nk-1} x_0 \bmod m \tag{6.28}$$

if the seed of the arbitrary sequence is $x_{nk}, x_{nk+1}, \ldots, x_{nk-1}$. This is particularly convenient for parallel computers where one computer generates

the seeds for the subsequences and the processing nodes, independently, generated the subsequences.

6.5.2 Leapfrogging

Another and probably more popular technique is leapfrogging. Let's consider one sequence of pseudo random numbers:

$$x_0, x_1, \ldots, x_k, x_{k+1}, \ldots, x_{2k}, x_{2k+1}, \ldots, x_{3k}, x_{3k+1}, \ldots, \tag{6.29}$$

One can break it into subsequences of k numbers:

$$x_0, x_k, x_{2k}, x_{3k}, \ldots \tag{6.30}$$

$$x_1, x_{1+k}, x_{1+2k}, x_{1+3k}, \ldots \tag{6.31}$$

$$x_2, x_{2+k}, x_{2+2k}, x_{2+3k}, \ldots \tag{6.32}$$

$$\ldots \tag{6.33}$$

The seeds $x_1, x_2, ..x_{k-1}$ are generated from x_0, and the independent sequences can be generated independently using the formula

$$x_{i+k} = a^k x_i \mathrm{mod} m \tag{6.34}$$

Therefore leapfrogging is a viable technique for parallel random number generators.

Here is an example of a usage of `leapfrog`:

Listing 6.3: in file: `nlib.py`

```
def leapfrog(mcg,k):
    a = mcg.a**k % mcg.m
    return [MCG(mcg.next(),a,mcg.m) for i in range(k)]
```

Here is an example of usage:

```
>>> generators=leapfrog(MCG(m),3)
>>> for k in xrange(3):
...     for i in xrange(5):
...         x=generators[k].random()
...         print k,'\t',i,'\t',x
```

The Mersenne twister algorithm implemented in `os.random` has leapfrogging built in. In fact, the module includes a `random.jumpahead(n)` method that allows us to efficiently skip n numbers.

6.5.3 Lehmer trees

Lehmer trees are binary trees, generated recursively, where each node contains a random number. We start from the root containing the seed, x_0, and we append two children containing, respectively,

$$x_i^L = (a_L x_{i-1} + c_L) \mathrm{mod} m \tag{6.35}$$

$$x_i^R = (a_R x_{i-1} + c_R) \mathrm{mod} m \tag{6.36}$$

then, recursively, append nodes to the children.

6.6 Generating random numbers from a given distribution

In this section and the next, we provide examples of distributions other than uniform and algorithms to generate numbers using these distributions. The general strategy consists of finding ways to map uniform random numbers into numbers following a different distribution. There are two general techniques for mapping uniform into nonuniform random numbers:

- accept–reject (applies to both discrete and continuous distributions)
- inversion methods (applies to continuous distributions only)

Consider the problem of generating a random number x from a given distribution $p(x)$. The *accept–reject method* consists of generating x using a different distribution, $g(x)$, and a uniform random number, u, between 0,1. If $u < p(x)/Mg(x)$ (M is the max of $p(x)/g(x)$), then x is the desired random number following distribution $p(x)$. If not, try another number.

To visualize why this works, imagine graphing the distribution p of the random variable x onto a large rectangular board and throwing darts at it, the coordinates of the dart being (x, u). Assume that the darts are uniformly distributed around the board. Now take off (reject) all of the darts that are outside the curve. The remaining darts will be distributed uniformly within the curve, and the x-positions of these darts will be distributed according to the random variable's density. This is because

there is the most room for the darts to land where the curve is highest and thus the probability density is greatest.

The g distribution is nothing but a shape so that all darts we throw are below it. There are two particular cases. In one case, $g = p$, we only throw darts below the p that we want; therefore we accept them all. This is the most efficient case, but it is not of practical interest because it means the accept–reject is not doing anything, as we already know now to generate numbers according to p. The other case is $g(x) = constant$. This means we generate the x uniformly before the accept–reject. This is equivalent to throwing the darts everywhere on the square board, without even trying to be below the curve p.

The inversion method instead is more efficient but requires some math. It states that if $F(x)$ is a cumulative distribution function and u is a uniform random number between 0 and 1, then $x = F^{-1}(u)$ is a random number with distribution $p(x) = F'(x)$. For those distributions where F can be expressed in analytical terms and inverted, the inversion method is the best way to generate random numbers. An example is the exponential distribution.

We will create a new class `RandomSource` that includes methods to generate the random number.

6.6.1 Uniform distribution

The uniform distributions are simple probability distributions which, in the discrete case, can be characterized by saying that all possible values are equally probable. In the continuous case, one says that all intervals of the same length are equally probable.

There are two types of uniform distribution: discrete and continuous.

Here we consider the discrete case as we implement it into a `randint` method:

Listing 6.4: in file: `nlib.py`

```
1 class RandomSource(object):
```

```
    def __init__(self,generator=None):
        if not generator:
            import random as generator
        self.generator = generator
    def random(self):
        return self.generator.random()
    def randint(self,a,b):
        return int(a+(b-a+1)*self.random())
```

Notice that the random `RandomSource` constructor expects a generator such as `MCG`, `MersenneTwister`, or simply `random` (default value). The `random()` method is a proxy method for the equivalent method of the underlying generator object.

We can use `randint` to generate a random choice from a finite set when each option has the same probability:

Listing 6.5: in file: `nlib.py`

```
    def choice(self,S):
        return S[self.randint(0,len(S)-1)]
```

6.6.2 Bernoulli distribution

The Bernoulli distribution, named after Swiss scientist James Bernoulli, is a discrete probability distribution that takes value 1 with probability of success p and value 0 with probability of failure $q = 1 - p$:

$$p(k) \equiv \begin{cases} p & \text{if } k = 1 \\ 1-p & \text{if } k = 0 \\ 0 & \text{if not } k \in \{0,1\} \end{cases} \tag{6.37}$$

A Bernoulli random variable has an expected value of p and variance of pq.

We implement it by adding a corresponding method to the `RandomSource` class:

Listing 6.6: in file: `nlib.py`

```
    def bernoulli(self,p):
        return 1 if self.random()<p else 0
```

6.6.3 Biased dice and table lookup

A generalization of the Bernoulli distribution is a distribution in which we have a finite set of choices, each with an associated probability. The table can be a list of tuples `(value, probability)` or a dictionary of `value:probability`:

Listing 6.7: in file: `nlib.py`

```
    def lookup(self,table, epsilon=1e-6):
        if isinstance(table,dict): table = table.items()
        u = self.random()
        for key,p in table:
            if u<p+epsilon:
                return key
            u = u - p
        raise ArithmeticError('invalid probability')
```

Let's say we want a random number generator that can only produce the outcomes 0, 1 or 2 with known probabilities:

$$\text{Prob}(X = 0) = 0.50 \tag{6.38}$$

$$\text{Prob}(X = 1) = 0.23 \tag{6.39}$$

$$\text{Prob}(X = 2) = 0.27 \tag{6.40}$$

Because the probability of the possible outcomes are rational numbers (fractions), we can proceed as follows:

```
>>> def test_lookup(nevents=100,table=[(0,0.50),(1,0.23),(2,0.27)]):
...     g = RandomSource()
...     f=[0,0,0]
...     for k in xrange(nevents):
...         p=g.lookup(table)
...         print p,
...         f[p]=f[p]+1
...     print
...     for i in xrange(len(table)):
...         f[i]=float(f[i])/nevents
...         print 'frequency[%i]=%f' % (i,f[i])
```

which produces the following output:

```
0 1 2 0 0 0 2 2 2 2 2 0 0 0 2 1 1 2 0 0 2 1 2 0 1
0 0 0 0 0 0 0 0 0 0 1 2 2 0 0 1 2 2 0 0 1 0 0 1 0
0 0 0 0 0 2 2 0 2 0 2 0 0 0 0 2 1 2 0 2 0 2 0 0 0
0 0 0 2 2 0 0 0 0 2 1 1 0 2 0 0 0 0 0 0 1 0 1 0 0 0
```

```
frequency[0]=0.600000
frequency[1]=0.140000
frequency[2]=0.260000
```

Eventually, by repeating the experiment many more times, the frequencies of 0,1 and 2 will approach the input probabilities.

Given the output frequencies, what is the probability that they are compatible with the input frequency? The answer to this question is given by the χ^2 and its distribution. We discuss this later in the chapter.

In some sense, we can think of the table lookup as an application of the linear search. We start with a segment of length 1, and we break it into smaller contiguous intervals of length $\text{Prob}(X = 0), \text{Prob}(X = 1), \ldots, \text{Prob}(x = n - 1)$ so that $\sum \text{Prob}(X = i) = 1$. We then generate a random point on the initial segment, and we ask in which of the n intervals it falls. The table lookup method linearly searches the interval.

This technique is $\Theta(n)$, where n is the number of outcomes of the computation. Therefore it becomes impractical if the number of cases is large. In this case, we adopt one of the two possible techniques: the Fishman–Yarberry method or the accept–reject method.

6.6.4 Fishman–Yarberry method

The Fishman–Yarberry [52] (F-Y) method is an improvement over the naive table lookup that runs in $O(\lceil \log_2 n \rceil)$. As the naive table lookup is an application of the linear search, the F-Y is an application of the binary search.

Let's assume that $n = 2^t$ is an exact power of 2. If this is not the case, we can always reduce to this case by adding new values to the lookup table corresponding to 0 probability. The basic data structure behind the F-Y method is an array of arrays a_{ij} built according to the following rules:

- $\forall j \geq 0, a_{0j} = \text{Prob}(X = x_j)$
- $\forall j \geq 0$ and $i > 0$, $a_{ij} = a_{i-1,2j} + a_{i-1,2j+1}$

Note that $0 \leq i < t$ and $\forall i \geq 0, 0 \leq j < 2^{t-i}$, where $t = \log_2 n$. The array

of arrays a can be represented as follows:

$$a_{ij} = \begin{pmatrix} a_{00} & a_{01} & a_{02} & \dots & a_{0,n-1} \\ \dots & \dots & \dots & \dots & \\ a_{t-2,0} & a_{t-2,1} & a_{t-2,2} & a_{t-2,3} & \\ a_{t-1,0} & a_{t-1,1} & & & \end{pmatrix} \tag{6.41}$$

In other words, we can say that

- a_{ij} represents the probability

$$\text{Prob}(X = x_j) \tag{6.42}$$

- a_{1j} represents the probability

$$\text{Prob}(X = x_{2j} \text{ or } X = x_{2j+1}) \tag{6.43}$$

- a_{4j} represents the probability

$$\text{Prob}(X = x_{4j} \text{ or } X = x_{4j+1} \text{ or } X = x_{4j+2} \text{ or } X = x_{4j+3}) \tag{6.44}$$

- a_{ij} represents the probability

$$\text{Prob}(X \in \{x_k | 2^i j \leq k < 2^i(j+1)\}) \tag{6.45}$$

This algorithm works like the binary search, and at each step, it confronts the uniform random number u with a_{ij} and decides if u falls in the range $\{x_k | 2^i j \leq k < 2^i(j+1)\}$ or in the complementary range $\{x_k | 2^i(j+1) \leq k < 2^i(j+2)\}$ and decreases i.

Here is the algorithm implemented as a class member function. The constructor of the class creates an array a once and for all. The method discrete_map maps a uniform random number u into the desired discrete integer:

```
class FishmanYarberry(object):
    def __init__(self,table=[[0,0.2], [1,0.5], [2,0.3]]):
        t=log(len(table),2)
        while t!=int(t):
            table.append([0,0.0])
```

```
            t=log(len(table),2)
        t=int(t)
        a=[]
        for i in xrange(t):
            a.append([])
            if i==0:
                for j in xrange(2**t):
                    a[i].append(table[j,1])
            else:
                for j in xrange(2**(t-i)):
                    a[i].append(a[i-1][2*j]+a[i-1][2*j+1])
        self.table=table
        self.t=t
        self.a=a

    def discrete_map(self, u):
        i=int(self.t)-1
        j=0
        b=0
        while i>0:
            if u>b+self.a[i][j]:
                b=b+self.a[i][j]
                j=2*j+2
            else:
                j=2*j
            i=i-1
        if u>b+self.a[i][j]:
            j=j+1
        return self.table[j][0]
```

6.6.5 Binomial distribution

The binomial distribution is a discrete probability distribution that describes the number of successes in a sequence of n independent experiments, each of which yields success with probability p. Such a success–failure experiment is also called a Bernoulli experiment.

A typical example is the following: 7% of the population are left-handed. You pick 500 people randomly. How likely is it that you get 30 or more left-handed? The number of left-handed you pick is a random variable X that follows a binomial distribution with $n = 500$ and $p = 0.07$. We are interested in the probability $\text{Prob}(X = 30)$.

In general, if the random variable X follows the binomial distribution

with parameters n and p, the probability of getting exactly k successes is given by

$$p(k) = \text{Prob}(X = k) \equiv \binom{n}{k} p^k (1-p)^{n-k} \tag{6.46}$$

for $k = 0, 1, 2, \ldots, n$.

The formula can be understood as follows: we want k successes (p^k) and $n - k$ failures ($(1-p)^{n-k}$). However, the k successes can occur anywhere among the n trials, and there are $\binom{n}{k}$ different ways of distributing k successes in a sequence of n trials.

The mean is $\mu_X = np$, and the variance is $\sigma_X^2 = np(1-p)$.

If X and Y are independent binomial variables, then $X + Y$ is again a binomial variable; its distribution is

$$p(k) = \text{Prob}(X = k) = \binom{n_X + n_Y}{k} p^k (1-p)^{n-k} \tag{6.47}$$

We can generate random numbers following binomial distribution using a table lookup with table

$$\text{table}[k] = \text{Prob}(X = k) = \binom{n}{k} p^k (1-p)^{n-k} \tag{6.48}$$

For large n, it may be convenient to avoid storing the table and use the formula directly to compute its elements on a need-to-know basis. Moreover, because the table is accessed sequentially by the table lookup algorithm, one may just notice that the current recursive relation holds:

$$\text{Prob}(X = 0) = (1-p)^n \tag{6.49}$$

$$\text{Prob}(X = k+1) = \frac{n}{k+1} \frac{p}{1-p} \text{Prob}(X = k) \tag{6.50}$$

This allows for a very efficient implementation:

Listing 6.8: in file: `nlib.py`

```
1    def binomial(self,n,p,epsilon=1e-6):
```

```
        u = self.random()
        q = (1.0-p)**n
        for k in xrange(n+1):
            if u<q+epsilon:
                return k
            u = u - q
            q = q*(n-k)/(k+1)*p/(1.0-p)
        raise ArithmeticError('invalid probability')
```

6.6.6 Negative binomial distribution

In probability theory, the negative binomial distribution is the probability distribution of the number of trials n needed to get a fixed (nonrandom) number of successes k in a Bernoulli process. If the random variable X is the number of trials needed to get r successes in a series of trials where each trial has probability of success p, then X follows the negative binomial distribution with parameters r and p:

$$p(n) = \text{Prob}(X = n) = \binom{n-1}{k-1} p^k (1-p)^{n-k} \tag{6.51}$$

Here is an example:

John, a kid, is required to sell candy bars in his neighborhood to raise money for a field trip. There are thirty homes in his neighborhood, and he is told not to return home until he has sold five candy bars. So the boy goes door to door, selling candy bars. At each home he visits, he has a 0.4 probability of selling one candy bar and a 0.6 probability of selling nothing.

- What's the probability of selling the last candy bar at the nth house?

$$p(n) = \binom{n-1}{4} 0.4^5 0.6^{n-5} \tag{6.52}$$

- What's the probability that he finishes on the tenth house?

$$p(10) = \binom{9}{4} 0.4^5 0.6^5 = 0.10 \tag{6.53}$$

- What's the probability that he finishes on or before reaching the eighth house? Answer: To finish on or before the eighth house, he must finish at the fifth, sixth, seventh, or eighth house. Sum those probabilities:

$$\sum_{i=5,6,7,8} p(i) = 0.1737 \tag{6.54}$$

- What's the probability that he exhausts all houses in the neighborhood without selling the five candy bars?

$$1 - \sum_{i=5,..,30} p(i) = 0.0015 \tag{6.55}$$

As we the binomial distribution, we can find an efficient recursive formula for the negative binomial distribution:

$$\text{Prob}(X = k) \quad = p^k \tag{6.56}$$

$$\text{Prob}(X = n+1) = \frac{n}{n-k+1}(1-p)\text{Prob}(X = n) \tag{6.57}$$

This allows for a very efficient implementation:

Listing 6.9: in file: `nlib.py`

```
    def negative_binomial(self,k,p,epsilon=1e-6):
        u = self.random()
        n = k
        q = p**k
        while True:
            if u<q+epsilon:
                return n
            u = u - q
            q = q*n/(n-k+1)*(1-p)
            n = n + 1
        raise ArithmeticError('invalid probability')
```

Notice once again that, unlike the binomial case, here k is fixed, not n, and the random variable has a minimum value of k but no upper bound.

6.6.7 Poisson distribution

The Poisson distribution is a discrete probability distribution discovered by Siméon-Denis Poisson. It describes a random variable X that counts, among other things, the number of discrete occurrences (sometimes called *arrivals*) that take place during a time interval of given length. The probability that there are exactly x occurrences (x being a natural number including 0, $k = 0, 1, 2, \ldots$) is

$$p(k) = \text{Prob}(X = k) = e^{-\lambda}\frac{\lambda^k}{k!} \tag{6.58}$$

The Poisson distribution arises in connection with Poisson processes. It applies to various phenomena of discrete nature (i.e., those that may happen 0, 1, 2, 3,..., times during a given period of time or in a given area) whenever the probability of the phenomenon happening is constant in time or space. The Poisson distribution differs from the other distributions considered in this chapter because it is different than zero for any natural number k rather than for a finite set of k values.

Examples include the following:

- The number of unstable nuclei that decayed within a given period of time in a piece of radioactive substance.
- The number of cars that pass through a certain point on a road during a given period of time.
- The number of spelling mistakes a secretary makes while typing a single page.
- The number of phone calls you get per day.
- The number of times your web server is accessed per minute.
- The number of roadkill you find per mile of road.
- The number of mutations in a given stretch of DNA after a certain amount of radiation.
- The number of pine trees per square mile of mixed forest.

- The number of stars in a given volume of space.

The limit of the binomial distribution with parameters n and $p = \lambda/n$, for n approaching infinity, is the Poisson distribution:

$$\frac{n!}{(n-k)!k!}\left(\frac{\lambda}{n}\right)^k\left(1-\frac{\lambda}{n}\right)^{n-k} \simeq e^{-\lambda}\frac{\lambda^k}{k!} + O(\frac{1}{n}) \tag{6.59}$$

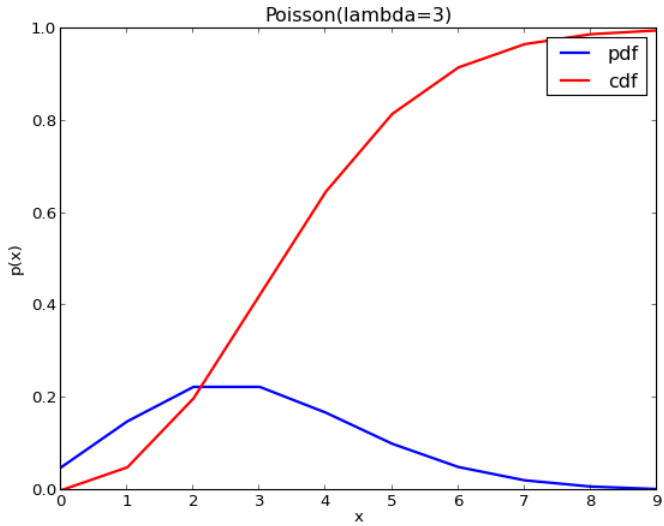

Figure 6.2: Example of Poisson distribution.

Intuitively, the meaning of λ is the following:

Let's consider a unitary time interval T and divide it into n subintervals of the same size. Let p_n be the probability of one success occurring in a single subinterval. For T fixed when $n \to \infty$, $p_n \to 0$ but the limit

$$\lim_{n\to\infty} pn \tag{6.60}$$

is finite. This limit is λ.

We can use the same technique adopted for the binomial distribution and

observe that for Poisson,

$$\text{Prob}(X = 0) \qquad = e^{-\lambda} \tag{6.61}$$

$$\text{Prob}(X = k+1) = \frac{\lambda}{k+1}\text{Prob}(X = k) \tag{6.62}$$

therefore the preceding algorithm can be modified into

Listing 6.10: in file: `nlib.py`

```
    def poisson(self,lamb,epsilon=1e-6):
        u = self.random()
        q = exp(-lamb)
        k=0
        while True:
            if u<q+epsilon:
                return k
            u = u - q
            q = q*lamb/(k+1)
            k = k+1
        raise ArithmeticError('invalid probability')
```

Note how this algorithm may take an arbitrary amount of time to generate a Poisson distributed random number, but eventually it stops. If u is very close to 1, it is possible that errors due to finite machine precision cause the algorithm to enter into an infinite loop. The $+\varepsilon$ term can be used to correct this unwanted behavior by choosing ε relatively small compared with the precision required in the computation, but larger than machine precision.

6.7 Probability distributions for continuous random variables

6.7.1 Uniform in range

A typical problem is generating random integers in a given range $[a, b]$, including the extreme. We can map uniform random numbers $y_i \in (0, 1)$ into integers by using the formula

$$h_i = a + \lfloor (b - a + 1)y_i \rfloor \tag{6.63}$$

Listing 6.11: in file: nlib.py

```
    def uniform(self,a,b):
        return a+(b-a)*self.random()
```

6.7.2 Exponential distribution

The exponential distribution is used to model Poisson processes, which are situations in which an object initially in state A can change to state B with constant probability per unit time λ. The time at which the state actually changes is described by an exponential random variable with parameter λ. Therefore the integral from 0 to T over $p(t)$ is the probability that the object is in state B at time T.

The probability mass function is given by

$$p(x) = \lambda e^{-\lambda x} \tag{6.64}$$

The exponential distribution may be viewed as a continuous counterpart of the geometric distribution, which describes the number of Bernoulli trials necessary for a discrete process to change state. In contrast, the exponential distribution describes the time for a continuous process to change state.

Examples of variables that are approximately exponentially distributed are as follows:

- the time until you have your next car accident
- the time until you get your next phone call
- the distance between mutations on a DNA strand
- the distance between roadkill

An important property of the exponential distribution is that it is memoryless: the chance that an event will occur s seconds from now does not depend on the past. In particular, it does not depend on how much time

we have been waiting already. In a formula we can write this condition as

$$\text{Prob}(X > s + t | X > t) = \text{Prob}(X > s) \tag{6.65}$$

Any process satisfying the preceding condition is a Poisson process. The number of events per time unit is given by the Poisson distribution, and the time interval between consecutive events is described by the exponential distribution.

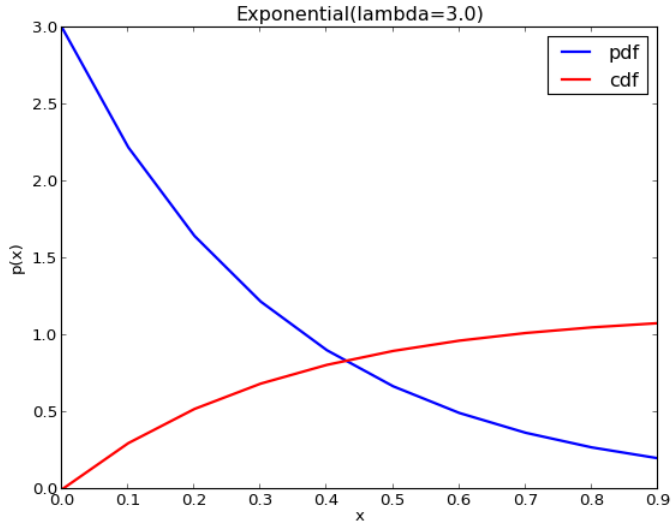

Figure 6.3: Example of exponential distribution.

The exponential distribution can be generated using the inversion method. The scope is to determine a function $x = f(u)$ that maps a uniformly distributed variable u into a continuous random variable x with probability mass function $p(x) = \lambda e^{-\lambda x}$.

According to the inversion method, we proceed by computing F:

$$F(x) = \int_0^x p(y)dy = 1 - e^{-\lambda x} \tag{6.66}$$

and we then invert $u = F(x)$, thus obtaining

$$x = -\frac{1}{\lambda}\log(1-u) \tag{6.67}$$

Now notice that if u is uniform, $1-u$ is also uniform; therefore we can further simplify:

$$x = -\frac{1}{\lambda}\log u \tag{6.68}$$

We implement as follows:

Listing 6.12: in file: `nlib.py`

```
    def exponential(self,lamb):
        return -log(self.random())/lamb
```

This is an important distribution, and Python has a function for it:

```
random.expovariate(lamb)
```

6.7.3 Normal/Gaussian distribution

The normal distribution (also known as *Gaussian distribution*) is an extremely important probability distribution considered in statistics. Here is the probability mass function:

$$p(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \tag{6.69}$$

where $E[X] = \mu$ and $E[(x-\mu)^2] = \sigma^2$.

The standard normal distribution is the normal distribution with a mean of 0 and a standard deviation of 1. Because the graph of its probability density resembles a bell, it is often called the *bell curve*.

The Gaussian distribution has two important properties:

- The average of many independent random variables with finite mean and finite variance tends to be a Gaussian distribution.
- The sum of two independent Gaussian random variables with means μ_1 and μ_2 and variances σ_1^2 and σ_2^2 is also a Gaussian random variable with mean $\mu = \mu_1 + \mu_2$ and variance $\sigma^2 = \sigma_1^2 + \sigma_2^2$.

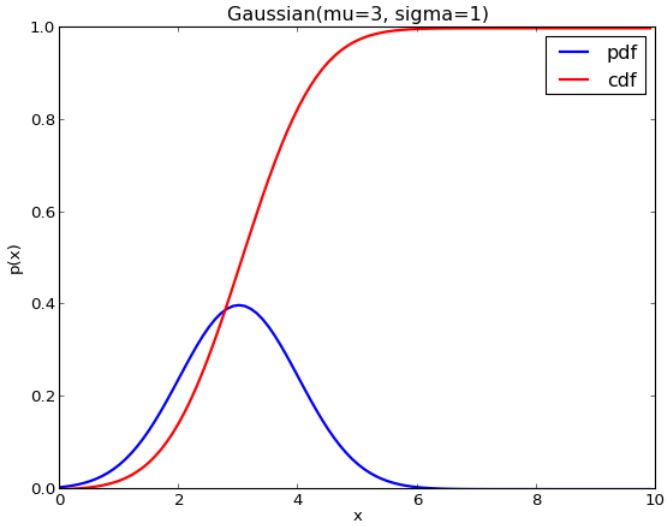

Figure 6.4: Example of Gaussian distribution.

There is no way to map a uniform random number into a Gaussian number but there is an algorithm to generate two independent Gaussian random numbers (y_1 and y_2) using two independent uniform random numbers (x_1 and x_2):

- computing $v_1 = 2x_1 - 1, v_2 = 2x_2 - 1$ and $s = v_1^2 + v_2^2$
- if $s > 1$ start again
- $y_1 = v_1\sqrt{(-2/s)\log s}$ and $y_2 = v_2\sqrt{(-2/s)\log s}$

Listing 6.13: in file: `nlib.py`

```
    def gauss(self,mu=0.0,sigma=1.0):
        if hasattr(self,'other') and self.other:
            this, other = self.other, None
        else:
            while True:
                v1 = self.random(-1,1)
                v2 = self.random(-1,1)
                r = v1*v1+v2*v2
                if r<1: break
            this = sqrt(-2.0*log(r)/r)*v1
```

```
            self.other = sqrt(-2.0*log(r)/r)*v1
        return mu+sigma*this
```

Note how the first time the method next is called, it generates two Gaussian numbers (*this* and *other*), stores *other*, and returns *this*. Every other time, the method next is called if *other* is stored, and it returns a number; otherwise it recomputes *this* and *other* again.

To map a random Gaussian number y with mean 0 and standard deviation 1 into another Gaussian number y' with mean μ and standard deviation σ,

$$y' = \mu + y\sigma \tag{6.70}$$

We used this relation in the last line of the code.

This is also an important distribution, and Python has a function for it:

```
random.gauss(mu,sigma)
```

Given a Gaussian random variable with mean μ and standard deviation σ, it is often useful to know how many standard deviations a correspond to a confidence c defined as

$$c = \int_{\mu-a\sigma}^{\mu+a\sigma} p(x)\mathrm{d}x \tag{6.71}$$

The following algorithm generates a table of a versus c given μ and σ:

Listing 6.14: in file: `nlib.py`

```
def confidence_intervals(mu,sigma):
    """Computes the normal confidence intervals"""
    CONFIDENCE=[
        (0.68,1.0),
        (0.80,1.281551565545),
        (0.90,1.644853626951),
        (0.95,1.959963984540),
        (0.98,2.326347874041),
        (0.99,2.575829303549),
        (0.995,2.807033768344),
        (0.998,3.090232306168),
        (0.999,3.290526731492),
        (0.9999,3.890591886413),
        (0.99999,4.417173413469)
        ]
    return [(a,mu-b*sigma,mu+b*sigma) for (a,b) in CONFIDENCE]
```

6.7.4 Pareto distribution

The Pareto distribution, named after the economist Vilfredo Pareto, is a power law probability distribution that coincides with social, scientific, geophysical, actuarial, and many other types of observable phenomena. Outside the field of economics, it is sometimes referred to as the Bradford distribution. Its cumulative distribution function is

$$F(x) \equiv \text{Prob}(X < x) = 1 - \left(\frac{x_m}{x}\right)^{\alpha} \tag{6.72}$$

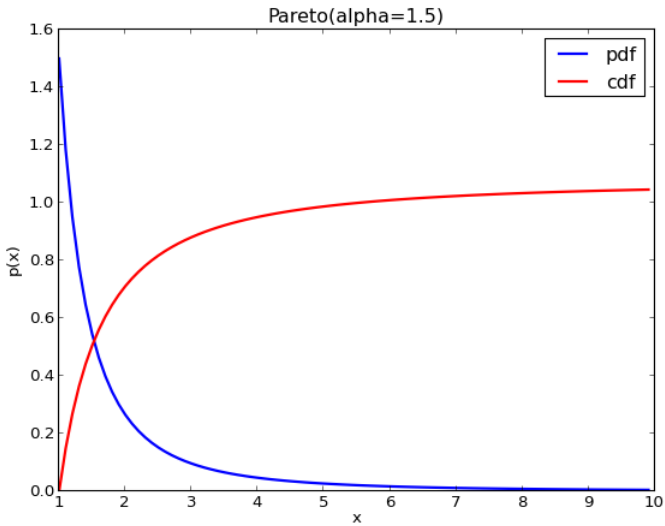

Figure 6.5: Example of Pareto distribution.

It can be implemented as follows using the inversion method:

Listing 6.15: in file: `nlib.py`

```
    def pareto(self,alpha,xm):
        u = self.random()
        return xm*(1.0-u)**(-1.0/alpha)
```

The Python function to generate Pareto distributed random numbers is

```
xm * random.paretovariate(alpha)
```

The Pareto distribution is an example of Levy distribution. The Central Limit theorem does not apply to it.

6.7.5 In and on a circle

We can generate a random point (x, y) uniformly distributed on a circle by generating a random angle.

$$x = \cos(2\pi u) \tag{6.73}$$

$$y = \sin(2\pi u) \tag{6.74}$$

Listing 6.16: in file: `nlib.py`

```
    def point_on_circle(self, radius=1.0):
        angle = 2.0*pi*self.random()
        return radius*math.cos(angle), radius*math.sin(angle)
```

We can generate a random point uniformly distributed inside a circle by generating, independently, the x and y coordinates of points inside a square and rejecting those outside the circle:

Listing 6.17: in file: `nlib.py`

```
    def point_in_circle(self,radius=1.0):
        while True:
            x = self.uniform(-radius,radius)
            y = self.uniform(-radius,radius)
            if x*x+y*y < radius*radius:
                return x,y
```

6.7.6 In and on a sphere

A random point (x, y, z) uniformly distributed on a sphere of radius 1 is obtained by generating three uniform random numbers u_1, u_2, u_3,; compute $v_i = 2u_i - 1$, and if $v_1^2 + v_2^2 + v_3^2 \leq 1$,

$$x = v_1 / \sqrt{v_1^2 + v_2^2 + v_3^2} \tag{6.75}$$

$$y = v_2 / \sqrt{v_1^2 + v_2^2 + v_3^2} \tag{6.76}$$

$$z = v_3 / \sqrt{v_1^2 + v_2^2 + v_3^2} \tag{6.77}$$

else start again.

Listing 6.18: in file: nlib.py

```
    def point_in_sphere(self,radius=1.0):
        while True:
            x = self.uniform(-radius,radius)
            y = self.uniform(-radius,radius)
            z = self.uniform(-radius,radius)
            if x*x+y*y*z*z < radius*radius:
                return x,y,z

    def point_on_sphere(self, radius=1.0):
        x,y,z = self.point_in_sphere(radius)
        norm = math.sqrt(x*x+y*y+z*z)
        return x/norm,y/norm,z/norm
```

6.8 Resampling

So far we always generated random numbers by modeling the random variable (e.g., uniform, or exponential, or Pareto) and using an algorithm to generate possible values of the random variables.

We now introduce a different methodology, which we will need later when talking about the bootstrap method. If we have a population S of equally distributed events and we need to generate an event from the same distribution as the population, we can simply draw a random element from the population. In Python, this is done with

```
>>> S = [1,2,3,4,5,6]
>>> print random.choice(S)
```

We can therefore generate a sample of random events by repeating this procedure. This is called *resampling* [53]:

Listing 6.19: in file: nlib.py

```
def resample(S,size=None):
    return [random.choice(S) for i in xrange(size or len(S))]
```

6.9 Binning

Binning is the process of dividing a space of possible events into partitions and counting how many events fall into each partition. We can bin the numbers generated by a pseudo random number generator and measure the distribution of the random numbers.

Let's consider the following program:

```
def bin(generator,nevents,a,b,nbins):
    # create empty bins
    bins=[]
    for k in xrange(nbins):
        bins.append(0)
    # fill the bins
    for i in xrange(nevents):
        x=generator.uniform()
        if x>=a and x<=b:
            k=int((x-a)/(b-a)*nbins)
            bins[k]=bins[k]+1
    # normalize bins
    for i in xrange(nbins):
        bins[i]=float(bins[i])/nevents
    return bins

def test_bin(nevents=1000,nbins=10):
    bins=bin(MCG(time()),nevents,0,1,nbins)
    for i in xrange(len(bins)):
        print i, bins[i]

>>> test_bin()
```

It produces the following output:

```
i frequency[i]
0 0.101
1 0.117
2 0.092
3 0.091
4 0.091
5 0.122
6 0.096
7 0.102
8 0.090
9 0.098
```

Note that

- all bins have the same size 1/nbins;
- the size of the bins is normalized, and the sum of the values is 1
- the distribution of the events into bins approaches the distribution of the numbers generated by the random number generator

As an experiment, we can do the same binning on a larger number of events,

```
>>> test_bin(100000)
```

which produces the following output:

```
i frequency[i]
0 0.09926
1 0.09772
2 0.10061
3 0.09894
4 0.10097
5 0.09997
6 0.10056
7 0.09976
8 0.10201
9 0.10020
```

Note that these frequencies differ from 0.1 for less than 3%, whereas some of the preceding numbers differ from 0.11 for more than 20%.

7
Monte Carlo Simulations

7.1 Introduction

Monte Carlo methods are a class of algorithms that rely on repeated random sampling to compute their results, which are otherwise deterministic.

7.1.1 Computing π

The standard way to compute π is by applying the definition: π is the length of a semicircle with a radius equal to 1. From the definition, one can derive an exact formula:

$$\pi = 4 \arctan 1 \tag{7.1}$$

The arctan has the following Taylor series expansion:[1]:

$$\arctan x = \sum_{i=0} (-1)^i \frac{x^{2i+1}}{2i+1} \tag{7.8}$$

[1] Taylor expansion:

$$f(x) = f(0) + f'(0)x + \frac{1}{2!}f''(0)x^2 + \cdots + \frac{1}{i!}f^{(i)}(0)x^2 + \ldots. \tag{7.2}$$

and one can approximate π to arbitrary precision by computing the sum

$$\pi = \sum_{i=0} (-1)^i \frac{4}{2i+1} \tag{7.9}$$

We can use the program

```
def pi_Taylor(n):
    pi=0
    for i in xrange(n):
        pi=pi+4.0/(2*i+1)*(-1)**i
        print i,pi

>>> pi_Taylor(1000)
```

which produces the following output:

```
0 4.0
1 2.66666...
2 3.46666...
3 2.89523...
4 3.33968...
...
999 3.14..
```

A better formula is due to Plauffe,

$$\pi = \sum_{i=0} \frac{1}{16^i} \left(\frac{4}{8i+1} - \frac{2}{8i+4} - \frac{1}{8i+5} - \frac{1}{8i+6} \right) \tag{7.10}$$

which we can implement as follows: we can use the program

```
from decimal import Decimal
def pi_Plauffe(n):
```

---

and if $f(x) = \arctan x$ then:

$$f'(x) = \frac{d \arctan x}{dx} = \frac{1}{1+x^2} \to f'(0) = 1 \tag{7.3}$$

$$f''(x) = \frac{d^2 \arctan x}{d^2 x} = \frac{d}{dx} \frac{1}{1+x^2} = -\frac{2x}{(1+x^2)^2} \tag{7.4}$$

$$\ldots = \ldots \tag{7.5}$$

$$f^{(2i+1)}(x) = (-1)^i \frac{(2i)!}{(1+x^2)^{2i+1}} + x \cdots \to f^{(2i+1)}(0) = (-1)(2i)! \tag{7.6}$$

$$f^{(2i)}(x) = x \cdots \to f^{(2i)}(0) = 0 \tag{7.7}$$

```
    pi=Decimal(0)
    a,b,c,d = Decimal(4),Decimal(2),Decimal(1),Decimal(1)/Decimal(16)
    for i in xrange(n):
        i8 = Decimal(8)*i
        pi=pi+(d**i)*(a/(i8+1)-b/(i8+4)-c/(i8+5)-c/(i8+6))
    return pi
>>> pi_Plauffe(1000)
```

The preceding formula works and converges very fast and already in 100 iterations produces

$$\pi = 3.14159265358979323846264 33\ldots \tag{7.11}$$

There is a different approach based on the fact that π is also the area of a circle of radius 1. We can draw a square or area containing a quarter of a circle of radius 1. We can randomly generate points (x, y) with uniform distribution inside the square and check if the points fall inside the circle. The ratio between the number of points that fall in the circle over the total number of points is proportional to the area of the quarter of a circle ($\pi/4$) divided by the area of the square (1).

Here is a program that implements this strategy:

```
from random import *

def pi_mc(n):
    pi=0
    counter=0
    for i in xrange(n):
        x=random()
        y=random()
        if x**2 + y**2 < 1:
            counter=counter+1
        pi=4.0*float(counter)/(i+1)
        print i,pi

pi_mc(1000)
```

The output of the algorithm is shown in fig. 7.1.1.

The convergence rate in this case is very slow, and this algorithm is of no practical value, but the methodology is sound, and for some problems, this method is the only one feasible.

Let's summarize what we have done: we have formulated our problem

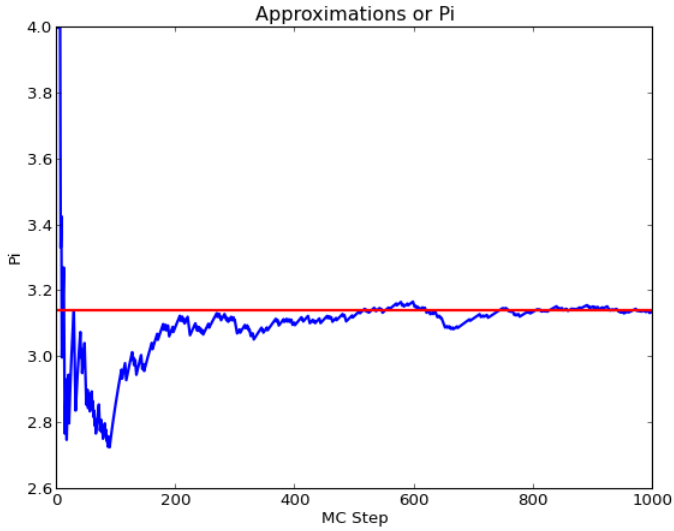

Figure 7.1: Convergence of **pi_mc**.

(compute π) as the problem of computing an area (the area of a quarter of a circle), and we have computed the area using random numbers. This is a particular example of a more general technique known as a Monte Carlo integration. In fact, the computation of an area is equivalent to the problem of computing an integral.

Sometimes the formula is not known, or it is too complex to compute reliably, hence a Monte Carlo solution becomes preferable.

7.1.2 Simulating an online merchant

Let's consider an online merchant. A website is visited many times a day. From the logfile of the web application, we determine that the average number of visitors in a day is 976, the number of visitors is Gaussian distributed, and the standard deviation is 352. We also observe that each visitor has a 5% probability of purchasing an item if the item is in stock and a 2% probability to buy an item if the item is not in stock.

The merchant sells only one type of item that costs $100 per unit. The merchant maintains N items in stock. The merchant pays $30 a day to store each unit item in stock. What is the optimal N to maximize the average daily income of the merchant?

This problem cannot easily be formulated analytically or reduced to the computation of an integral, but it can easily be simulated.

In particular, we simulate many days and, for each day i, we start with N items in stock, and we loop over each simulated visitor. If the visitor finds an item in stock, he buys it with a 5% probability (producing an income of $70), whereas if the item is not in stock, he buys it with 2% probability (producing an income of $100). At the end of each day, we pay $30 for each item remaining in stock.

Here is a program that takes N (the number of items in stock) and d (the number of simulated days) and computes the average daily income:

```
def simulate_once(N):
    profit = 0
    loss = 30*N
    instock = N
    for j in xrange(int(gauss(976,352))):
        if instock>0:
            if random()<0.05:
                instock=instock-1
                profit = profit + 100
        else:
            if random()<0.02:
                profit = profit + 100
    return profit-loss

def simulate_many(N,ap=1,rp=0.01,ns=1000):
    s = 0.0
    for k in xrange(1,ns):
        x = simulate_once(N)
        s += x
        mu = s/k
        if k>10 and mu-mu_old<max(ap,rp*mu):
            return mu
        else:
            mu_old = mu
    raise ArithmeticError('no convergence')
```

By looping over different N (items in stock), we can compute the average

daily income as a function of N:

```
>>> for N in xrange(0,100,10):
>>>     print N,simulate_many(N,ap=100)
```

The program produces the following output:

```
n income
0 1955
10 2220
20 2529
30 2736
40 2838
50 2975
60 2944
70 2711
80 2327
90 2178
```

From this we deduce that the optimal number of items to carry in stock is about 50. We could increase the resolution and precision of the simulation by increasing the number of simulated days and reducing the step of the amount of items in stock.

Note that the statement `gauss(976,352)` generates a random floating point number with a Gaussian distribution centered at 976 and standard deviation equal to 352, whereas the statement

```
if random()<0.05:
```

ensures that the subsequent block is executed with a probability of 5%.

The basic ingredients of every Monte Carlo simulation are here: (1) a function that simulates the system once and uses random variables to model unknown quantities; (2) a function that repeats the simulation many times to compute an average.

Any Monte Carlo solver comprises the following parts:

- A generator of random numbers (such as we have discussed in the previous chapter)
- A function that uses the random number generator and can simulate the system once (we will call x the result of each simulate once)
- A function that calls the preceding simulation repeatedly and averages

the results until they converge $\mu = \frac{1}{N}\sum x_i$

- A function to estimate the accuracy of the result and determine when to stop the simulation, $\delta\mu <$ precision

7.2 Error analysis and the bootstrap method

The result of any MC computation is an average:

$$\mu = \frac{1}{N}\sum x_i \tag{7.12}$$

The error on this average can be estimated using the formula

$$\delta\mu = \frac{\sigma}{\sqrt{N}} = \sqrt{\frac{1}{N}\left(\frac{1}{N}\sum x_i^2 - \mu^2\right)} \tag{7.13}$$

This formula assumes the distribution of the x_i is Gaussian. Using this formula, we can compute a 68% confidence level for the MC computation of π, shown in fig. 7.2.

The purpose of the bootstrap [54] algorithm is computing the error in an average $\mu = (1/N)\sum x_i$ without making the assumption that the x_i are Gaussian.

The first step of the bootstrap methodology consists of computing the average not only on the initial sample $\{x_i\}$ but also on many data samples obtained by resampling the original data. If the number of elements N of the original sample were infinity, the average on each other sample would be the same. Because N is finite, each of these means produces slightly different results:

$$\mu_k = \frac{1}{N}\sum x_i^{[k]} \tag{7.14}$$

where $x_i^{[k]}$ is the ith element of resample k and μ_k is the average of that resample.

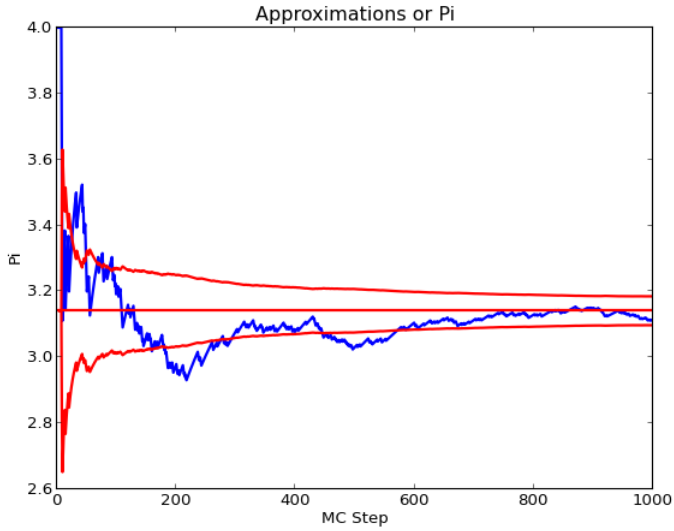

Figure 7.2: Convergence of π.

The second step of the bootstrap methodology consists of sorting the μ_k and finding two values μ^- and μ^+ that with a given percentage of the means follows in between those two values. The given percentage is the confidence level, and we set it to 68%.

Here is the complete algorithm:

Listing 7.1: in file: `nlib.py`

```
def bootstrap(x, confidence=0.68, nsamples=100):
    """Computes the bootstrap errors of the input list."""
    def mean(S): return float(sum(x for x in S))/len(S)
    means = [mean(resample(x)) for k in xrange(nsamples)]
    means.sort()
    left_tail = int(((1.0-confidence)/2)*nsamples)
    right_tail = nsamples-1-left_tail
    return means[left_tail], mean(x), means[right_tail]
```

Here is an example of usage:

```
>>> S = [random.gauss(2,1) for k in range(100)]
>>> print bootstrap(S)
(1.7767055865879007, 1.8968778392283303, 2.003420362236985)
```

In this example, the output consists of μ^-, μ, and μ^+.

Because S contains 100 random Gaussian numbers, with average 2 and standard deviation 1, we expect μ to be close to 2. We get 1.89. The bootstrap tells us that with 68% probability, the true average of these numbers is indeed between 1.77 and 2.00. The uncertainty $(2.00 - 1.77)/2 = 0.12$ is compatible with $\sigma/\sqrt{100} = 1/10 = 0.10$.

7.3 A general purpose Monte Carlo engine

We can now combine everything we have seen so far into a generic program that can be used to perform the most generic Monte Carlo computation/simulation:

Listing 7.2: in file: `nlib.py`

```
class MCEngine:
    """
    Monte Carlo Engine parent class.
    Runs a simulation many times and computes average and error in average.
    Must be extended to provide the simulate_once method
    """
    def simulate_once(self):
        raise NotImplementedError

    def simulate_many(self, ap=0.1, rp=0.1, ns=1000):
        self.results = []
        s1=s2=0.0
        self.convergence=False
        for k in xrange(1,ns):
            x = self.simulate_once()
            self.results.append(x)
            s1 += x
            s2 += x*x
            mu = float(s1)/k
            variance = float(s2)/k-mu*mu
            dmu = sqrt(variance/k)
            if k>10:
                if abs(dmu)<max(ap,abs(mu)*rp):
                    self.converence = True
                    break
        self.results.sort()
        return bootstrap(self.results)
```

The preceding class has two methods:

- `simulate_once` is not implemented because the class is designed to be subclassed, and the method is supposed to be implemented for each specific computation.
- `simulate_many` is the part that stays the same; it calls `simulate_once` repeatedly, computes average and error analysis, checks convergence, and computes bootstrap error for the result.

It is also useful to have a function, which we call **var** (aka *value at risk* [55]), which computes a numerical value so that the output of a given percentage of the simulations falls below that value:

Listing 7.3: in file: `nlib.py`

```
    def var(self, confidence=95):
        index = int(0.01*len(self.results)*confidence+0.999)
        if len(self.results)-index < 5:
            raise ArithmeticError('not enough data, not reliable')
        return self.results[index]
```

Now, as a first example, we can recompute π using this class:

```
>>> class PiSimulator(MCEngine):
...     def simulate_once(self):
...         return 4.0 if (random.random()**2+random.random()**2)<1 else 0.0
...
>>> s = PiSimulator()
>>> print s.simulate_many()
(2.1818181818181817, 2.909090909090909, 3.6363636363636362)
```

Our engine finds that the value of π with 68% confidence level is between 2.18 and 3.63, with the most likely value of 2.90. Of course, this is incorrect, because it generates too few samples, but the bounds are correct, and that is what matters.

7.3.1 Value at risk

Let's consider a business subject to random losses, for example, a large bank subject to theft from employees. Here we will make the following reasonable assumptions (which have been verified with data):

- There is no correlation between individual events.

- There is no correlation between the time when a loss event occurs and the amount of the loss.
- The time interval between losses is given by the exponential distribution (this is a Poisson process).
- The distribution of the loss amount is a Pareto distribution (there is a fat tail for large losses).
- The average number of losses is 10 per day.
- The minimum recorded loss is $5000. The average loss is $15,000.

Our goal is to simulate one year of losses and to determine

- The average total yearly loss
- How much to save to make sure that in 95% of the simulated scenarios, the losses can be covered without going broke

From these assumptions, we determine that the $\lambda = 10$ for the exponential distribution and $x_m = 3000$ for the Pareto distribution. The mean of the Pareto distribution is $\alpha x_m/(\alpha - 1) = 15,000$, from which we determine that $\alpha = 1.5$.

We can answer the first questions (the average total loss) simply multiplying the average number of losses per year, 52×5, by the number of losses in one day, 10, and by the average individual loss, $15,000, thus obtaining

$$[\text{average yearly loss}] = \$39,000,000 \tag{7.15}$$

To answer the second question, we would need to study the width of the distribution. The problem is that, for $\alpha = 1.5$, the standard deviation of the Pareto distribution is infinity, and analytical methods do not apply. We can do it using a Monte Carlo simulation:

Listing 7.4: in file: `risk.py`

```
from nlib import *
import random

class RiskEngine(MCEngine):
    def __init__(self,lamb,xm,alpha):
```

```
        self.lamb = lamb
        self.xm = xm
        self.alpha = alpha
    def simulate_once(self):
        total_loss = 0.0
        t = 0.0
        while t<260:
            dt = random.expovariate(self.lamb)
            amount = self.xm*random.paretovariate(self.alpha)
            t += dt
            total_loss += amount
        return total_loss

def main():
    s = RiskEngine(lamb=10, xm=5000, alpha=1.5)
    print s.simulate_many(rp=1e-4,ns=1000)
    print s.var(95)

main()
```

This produces the following output:

```
(38740147.179054834, 38896608.25084647, 39057683.35621854)
45705881.8776
```

The output of `simulate_many` should be compatible with the true result (defined as the result after an infinite number of iterations and at infinite precision) within the estimated statistical error.

The output of the `var` function answers our second questions: We have to save $45,705,881 to make sure that in 95% of cases our losses are covered by the savings.

7.3.2 Network reliability

Let's consider a network represented by a set of n_{nodes} nodes and n_{links} bidirectional links. Information packets travel on the network. They can originate at any node (`start`) and be addressed to any other node (`stop`). Each link of the network has a probability p of transmitting the packet (success) and a probability $(1-p)$ of dropping the packet (failure). The probability p is in general different for each link of the network.

We want to compute the probability that a packet starting in `start` finds a

successful path to reach `stop`. A path is successful if, for a given simulation, all links in the path succeed in carrying the packet.

The key trick in solving this problem is in finding the proper representation for the network. Since we are not requiring to determine the exact path but only proof of existence, we use the concept of equivalence classes.

We say that two nodes are in the same equivalence class if and only if there is a successful path that connects the two nodes.

The optimal data structure to implement equivalence classes is `DisjSets`, discussed in chapter 3.

To simulate the system, we create a class `Network` that extends `MCEngine`. It has a `simulate_once` method that tries to send a packet from `start` to `stop` and simulates the network once. During the simulation each link of the network may be up or down with given probability. If there is a path connecting the `start` node to the `stop` node in which all links of the network are up, than the packet transfer succeeds. We use the DisjointSets to represent sets of nodes connected together. If there is a link up connecting a node from a set to a node in another set, than the two sets are joined. If, in the end, the `start` and `stop` nodes are found to belong to the same set, then there is a path and `simulate_once` returns 1, otherwise it returns 0.

Listing 7.5: in file: `network.py`

```
from nlib import *
import random

class NetworkReliability(MCEngine):
    def __init__(self,n_nodes,start,stop):
        self.links = []
        self.n_nodes = n_nodes
        self.start = start
        self.stop = stop
    def add_link(self,i,j,failure_probability):
        self.links.append((i,j,failure_probability))
    def simulate_once(self):
        nodes = DisjointSets(self.n_nodes)
        for i,j,pf in self.links:
            if random.random()>pf:
                nodes.join(i,j)
        return nodes.joined(i,j)

```

```
def main():
    s = NetworkReliability(100,start=0,stop=1)
    for k in range(300):
        s.add_link(random.randint(0,99),
                   random.randint(0,99),
                   random.random())
    print s.simulate_many()

main()
```

7.3.3 Critical mass

Here we consider the simulation of a chain reaction in a fissile material, for example, the uranium in a nuclear reactor [56]. We assume a material is in a spherical shape of known radius. At each point there is a probability of a nuclear fission, which we model as the emission of two neutrons. Each of the two neutrons travels and hits an atom, thus causing another fission. The two neutrons are emitted in random opposite directions and travel a distance given by the exponential distribution. The new fissions may occur inside material itself or outside. If outside, they are ignored. If the number of fission events inside the material grows exponentially with time, we have a self-sustained chain reaction; otherwise, we do not.

Fig. 7.3.3 provides a representation of the process.

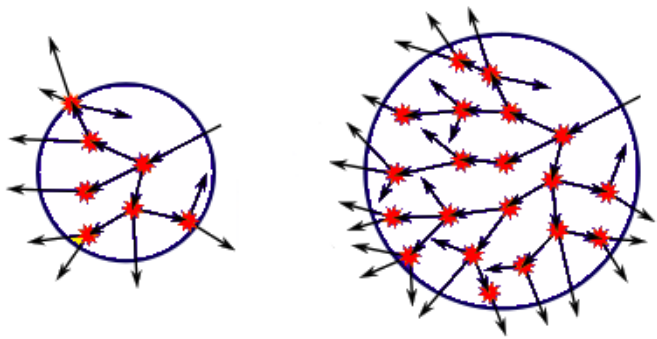

Figure 7.3: Example of chain reaction within a fissile material. If the mass is small, most of the decay products escape (left, sub-criticality), whereas if the mass exceeds a certain critical mass, there is a self-sustained chain reaction (right).

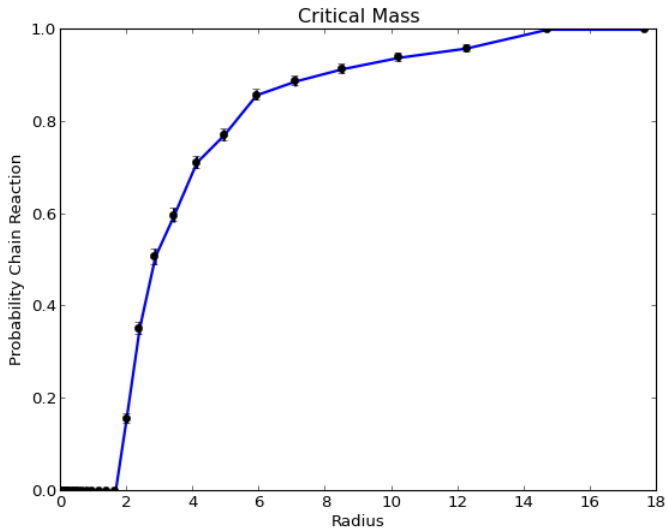

Figure 7.4: Probability of chain reaction in uranium.

Here is a possible implementation of the process. We store each event that happens inside the material in a queue (`events`). For each simulation, the queue starts with one event, and from it we generate two more, and so on. If the new events happen inside the material, we place the new events back in the queue. If the size of the queue shrinks to zero, then we are subcritical. If the size of the queue grows exponentially, we have a self-sustained chain reaction. We detect this by measuring the size of the queue and whether it exceeds a threshold (which we arbitrarily set to 200). The average free flight distance for a neutron in uranium is 1.91 cm. We use this number in our simulation. Given the radius of the material, `simulate_once` returns 1.0 if it detects a chain reaction and 0.0 if it does not. The output of `simulate_many` is the probability of a chain reaction:

Listing 7.6: in file: `nuclear.py`

```
from nlib import *
import math
import random

class NuclearReactor(MCEngine):
```

```
    def __init__(self,radius,mean_free_path=1.91,threshold=200):
        self.radius = radius
        self.density = 1.0/mean_free_path
        self.threshold = threshold
    def point_on_sphere(self):
        while True:
            x,y,z = random.random(), random.random(), random.random()
            d = math.sqrt(x*x+y*y+z*z)
            if d<1: return (x/d,y/d,z/d) # project on surface
    def simulate_once(self):
        p = (0,0,0)
        events = [p]
        while events:
            event = events.pop()
            v = self.point_on_sphere()
            d1 = random.expovariate(self.density)
            d2 = random.expovariate(self.density)
            p1 = (p[0]+v[0]*d1,p[1]+v[1]*d1,p[2]+v[2]*d1)
            p2 = (p[0]-v[0]*d2,p[1]-v[1]*d2,p[2]-v[2]*d2)
            if p1[0]**2+p1[1]**2+p1[2]**2 < self.radius:
                events.append(p1)
            if p2[0]**2+p2[1]**2+p2[2]**2 < self.radius:
                events.append(p2)
            if len(events) > self.threshold:
                return 1.0
        return 0.0

def main():
    s = NuclearReactor(MCEngine)
    data = []
    s.radius = 0.01
    while s.radius<21:
        r = s.simulate_many(ap=0.01,rp=0.01,ns=1000,nm=100)
        data.append((s.radius, r[1], (r[2]-r[0])/2))
        s.radius *= 1.2
    c = Canvas(title='Critical Mass',xlab='Radius',ylab='Probability Chain
         Reaction')
    c.plot(data).errorbar(data).save('nuclear.png')

main()
```

Fig. 7.3.3 shows the output of the program, the probability of a chain reaction as function of the size of the uranium mass. We find a critical radius between 2 cm and 10 cm, which corresponds to a critical mass between 0.5 kg and 60 kg. The official number is 15 kg for uranium 233 and 60 kg for uranium 235. The lesson to learn here is that it is not safe to accumulate too much fissile material together. This simulation can be

easily tweaked to determine the thickness of a container required to shield a radioactive material.

7.4 Monte Carlo integration

7.4.1 One-dimensional Monte Carlo integration

Let's consider a one-dimensional integral

$$I = \int_a^b f(x)dx \tag{7.16}$$

Let's now determine two functions $g(x)$ and $p(x)$ such that

$$p(x) = 0 \text{ for } x \in [-\infty, a] \cup [n, \infty] \tag{7.17}$$

and

$$\int_{-\infty}^{+\infty} p(x)dx = 1 \tag{7.18}$$

and

$$g(x) = f(x)/p(x) \tag{7.19}$$

We can interpret $p(x)$ as a probability mass function and

$$E[g(X)] = \int_{-\infty}^{+\infty} g(x)p(x)dx = \int_a^b f(x)dx = I \tag{7.20}$$

Therefore we can compute the integral by computing the expectation value of the function $g(X)$, where X is a random variable with a distribution (probability mass function) $p(x)$ different from zero in $[a, b]$ generated.

An obvious, although not in general an optimal choice, is

$$\begin{aligned} p(x) &\equiv \begin{cases} 1/(b-a) & \text{if } x \in [a, b] \\ 0 & \text{otherwise} \end{cases} \\ g(x) &\equiv (b-a)f(x) \end{aligned} \tag{7.21}$$

so that X is just a uniform random variable in $[a, b]$. Therefore

$$I = E\left[g(X)\right] = \frac{1}{N}\sum_{i=0}^{i<N} g(x_i) \tag{7.22}$$

This means that the integral can be evaluated by generating N random points x_i with uniform distribution in the domain, evaluating the integrand (the function f) on each point, averaging the results, and multiplying the average by the size of the domain $(b - a)$.

Naively, the error on the result can be estimated by computing the variance

$$\sigma^2 = \frac{1}{N}\sum_{i=0}^{i<N} \left[g(x_i) - \langle g\rangle\right]^2 \tag{7.23}$$

with

$$\langle g\rangle = \frac{1}{N}\sum_{i=0}^{i<N} g(x_i) \tag{7.24}$$

and the error on the result is given by:

$$\delta I = \sqrt{\frac{\sigma^2}{N}} \tag{7.25}$$

The larger the set of sample points N, the lower the variance and the error. The larger N, the better $E[g(X)]$ approximates the correct result I.

Here is a program in Python:

Listing 7.7: in file: `integrate.py`

```
class MCIntegrator(MCEngine):
    def __init__(self,f,a,b):
        self.f = f
        self.a = a
        self.b = b
    def simulate_once(self):
        a, b, f = self.a, self.b, self.f
        x = a+(b-a)*random.random()
        g = (b-a)*f(x)
        return g

def main():
    s = MCIntegrator(f lambda x: math.sin(x),a=0,b=1)
    print s.simulate_many()

main()
```

This technique is very general and can be extended to almost any integral assuming the integrand is smooth enough on the integration domain.

The choice (7.21) is not always optimal because the integrand may be very small in some regions of the integration domain and very large in other regions. Clearly some regions contribute more than others to the average, and one would like to generate points with a probability mass function that is as close as possible to the original integrand. Therefore we should choose a $p(x)$ according to the following conditions:

- $p(x)$ is very similar and proportional to $f(x)$
- given $F(x) = \int_{-\infty}^{x} p(x)dx$, $F^{-1}(x)$ can be computed analytically.

Any choice for $p(x)$ that makes the integration algorithm converge faster with less calls to `simulate_once` is called a *variance reduction technique*.

7.4.2 Two-dimensional Monte Carlo integration

The technique described earlier can easily be extended to two-dimensional integrals:

$$I = \int_{\mathfrak{D}} f(x_0, x_1)dx_0dx_1 \tag{7.26}$$

where $\mathfrak{D}$ is some two-dimensional domain. We determine two functions $g(x_0, x_1)$ and $p_0(x_0), p_1(x_1)$ such that

$$p_0(x_0) = 0 \text{ or } p_1(x_1) = 0 \text{ for } x \notin \mathfrak{D} \tag{7.27}$$

and

$$\int p_0(x_0)p_1(x_1)dx_0dx_1 = 1 \tag{7.28}$$

and

$$g(x_0, x_1) = \frac{f(x_0, x_1)}{p_0(x_0)p_1(x_1)} \tag{7.29}$$

We can interpret $p(x_0, x_1)$ as a probability mass function for two independent random variables X_0 and X_1 and

$$E[g(X_0, X_1)] = \int g(x_0, x_1)p_0(x_0)p_1(x_1)dx = \int_{\mathfrak{D}} f(x_0, x_1)dx_0dx_1 = I \tag{7.30}$$

Therefore

$$I = E[g(X_0, X_1)] = \frac{1}{N} \sum_{i=0}^{i<N} g(x_{i0}, x_{i1}) \tag{7.31}$$

7.4.3 *n*-dimensional Monte Carlo integration

The technique described earlier can also be extended to n-dimensional integrals

$$I = \int_{\mathfrak{D}} f(x_0, \ldots, x_{n-1}) dx_0 \ldots dx_{n-1} \tag{7.32}$$

where $\mathfrak{D}$ is some n-dimensional domain identified by a function $domain(x_0, \ldots, x_{n-1})$ equal to 1 if $\mathbf{x} = (x_0, \ldots, x_{n-1})$ is in the domain, 0 otherwise. We determine two functions $g(x_0, \ldots, x_{n-1})$ and $p(x_0, \ldots, x_{n-1})$ such that

$$p(x_0, \ldots, x_{n-1}) = 0 \text{ for } x \notin \mathfrak{D} \tag{7.33}$$

and

$$\int p(x_0, \ldots, x_{n-1}) dx_0 \ldots dx_{n-1} = 1 \tag{7.34}$$

and

$$g(x_0, \ldots, x_{n-1}) = f(x_0, \ldots, x_{n-1}) / p(x_0, \ldots, x_{n-1}) \tag{7.35}$$

We can interpret $p(x_0, \ldots, x_{n-1})$ as a probability mass function for n independent random variables $X_0 \ldots X_{n-1}$ and

$$E[g(X_0, \ldots, X_{n-1})] = \int g(x_0, \ldots, x_{n-1}) p(x_0, \ldots, x_{n-1}) dx \tag{7.36}$$

$$= \int_{\mathfrak{D}} f(x_0, \ldots, x_{n-1}) dx_0 \ldots dx_{n-1} = I \tag{7.37}$$

Therefore

$$I = E[g(X_0, .., X_{n-1})] = \frac{1}{N} \sum_{i=0}^{i<N} g(\mathbf{x}_i) \tag{7.38}$$

where for every point $\mathbf{x}_i$ is a tuple $(x_{i0}, x_{i1}, \ldots, x_{i,n-1})$.

As an example, we consider the integral

$$I = \int_0^1 dx_0 \int_0^1 dx_1 \int_0^1 dx_2 \int_0^1 dx_3 \sin(x_0 + x_1 + x_2 + x_3) \tag{7.39}$$

Here is the Python code:

```
>>> class MCIntegrator(MCEngine):
...     def simulate_once(self):
...         volume =  1.0
...         while True:
...             x = [random.random() for d in range(4)]
...             if sum(xi**2 for xi in x)<1: break
...         return volume*self.f(x)
>>> s = MCIntegrator()
>>> s.f = lambda x: math.sin(x[0]+x[1]+x[2]+x[3])
>>> print s.simulate_many()
```

7.5 Stochastic, Markov, Wiener, and processes

A *stochastic process* [57] is a random function, for example, a function that maps a variable n with domain D into X_n, where X_n is a random variable with domain R. In practical applications, the domain D over which the function is defined can be a time interval (and the stochastic is called a *time series*) or a region of space (and the stochastic process is called a *random field*). Familiar examples of time series include *random walks* [58]; stock market and exchange rate fluctuations; signals such as speech, audio, and video; or medical data such as a patient's EKG, EEG, blood pressure, or temperature. Examples of random fields include static images, random topographies (landscapes), or composition variations of an inhomogeneous material.

Let's consider a grasshopper moving on a straight line, and let X_n be the position of the grasshopper at time $t = n\Delta_t$. Let's also assume that at time 0, $X_0 = 0$. The position of the grasshopper at each future ($t > 0$) time is unknown. Therefore it is a random variable.

We can model the movements of the grasshopper as follows:

$$X_{n+1} = X_n + \mu + \varepsilon_n \Delta_x \tag{7.40}$$

where Δ_x is a fixed step and ε_n is a random variable whose distribution depends on the model; μ is a constant drift term (think of wind pushing the grasshopper in one direction). It is clear that X_{n+1} only depends on

X_n and ε_n; therefore the probability distribution of X_{n+1} only depends on X_n and the probability distribution of ε_n, but it does not depend on the past history of the grasshopper's movements at times $t < n\Delta_t$. We can write the statement by saying that

$$\text{Prob}(X_{n+1} = x | \{X_i\} \text{ for } i \leq n) = \text{Prob}(X_{n+1} = x | X_n) \tag{7.41}$$

A process in which the probability distribution of its future state only depends on the present state and not on the past is called a *Markov process* [59].

To complete our model, we need to make additional assumptions about the probability distribution of ε_n. We consider the two following cases:

- ε_n is a random variable with a Bernoulli distribution ($\varepsilon_n = +1$ with probability p and $\varepsilon_n = -1$ with probability $1 - p$).
- ε_n is a random variable with a normal (Gaussian) distribution with probability mass function $p(\varepsilon) = e^{-\varepsilon^2/2}$. Notice that the previous case (Bernoulli) is equivalent to this case (Gaussian) over long time intervals because the sum of many independent Bernoulli variables approaches a Gaussian distribution.

A continuous time stochastic process (when ε_n is a continuous random number) is called a *Wiener process* [60].

The specific case when ε_n is a Gaussian random variable is called an *Ito process* [61]. An Ito process is also a Wiener process.

7.5.1 Discrete random walk (Bernoulli process)

Here we assume a discrete random walk: ε_n equal to $+1$ with probability p and equal to -1 with probability $1 - p$. We consider discrete time intervals of equal length Δ_t; at each time step, if $\varepsilon_n = +1$, the grasshopper moves forward one unit (Δ_x) with probability p, and if $\varepsilon_n = -1$, he moves backward one unit ($-\Delta_x$) with probability $1 - p$.

For a total n steps, the probability of moving n_+ steps in a positive direc-

tion and $n_- = n - n_+$ in a negative direction is given by

$$\frac{n!}{n_+!(n-n_+)!} p^{n_+} (1-p)^{n-n_+} \tag{7.42}$$

The probability of going from $a = 0$ to $b = k\Delta_x > 0$ in a time $t = n\Delta_t > 0$ corresponds to the case when

$$n = n_+ + n_i \tag{7.43}$$

$$k = n_+ - n_- \tag{7.44}$$

that solved in n_+ gives $n_+ = (n+k)/2$, and therefore the probability of going from 0 to k in time $t = n\Delta_t$ is given by

$$\text{Prob}(n,k) = \frac{n!}{((n+k)/2)!((n-k)/2)!} p^{(n+k)/2} (1-p)^{(n-k)/2} \tag{7.45}$$

Note that $n + k$ has to be even, otherwise it is not possible for the grasshopper to reach $k\Delta_x$ in exactly n steps.

For large n, the following distribution in k/n tends to a Gaussian distribution.

7.5.2 Random walk: Ito process

Let's assume an Ito process for our random walk: ε_n is normally (Gaussian) distributed. We consider discrete time intervals of equal length Δ_t, at each time step if $\varepsilon_n = \varepsilon$ with probability mass function $p(\varepsilon) = e^{-\varepsilon^2/2}$. It turns out that eq.(7.40) gives

$$X_n = n\mu + \Delta_x \sum_{i=0}^{i<n} \varepsilon_i \tag{7.46}$$

Therefore the location of the random walker at time $t = n\Delta_t$ is given by the sum of n normal (Gaussian) random variables:

$$p(X_n) = \frac{1}{\sqrt{2\pi n\Delta_x^2}} e^{-(X_n - n\mu)^2/(2n\Delta_x^2)} \tag{7.47}$$

Notice how the mean and the variance of X_n are both proportional to n, whereas the standard deviation is proportional to $\sqrt{n}$.

$$\text{Prob}(a \le X_n \le b) = \frac{1}{\sqrt{2\pi n\Delta_x^2}} \int_a^b e^{-(X_n - n\mu)^2/(2n\Delta_x^2)} dx \tag{7.48}$$

$$= \frac{1}{\sqrt{2\pi}} \int_{(a-n\mu)/(\sqrt{n}\Delta_x)}^{(b-n\mu)/(\sqrt{n}\Delta_x)} e^{-x^2/2} dx \tag{7.49}$$

$$= \text{erf}(\frac{b - n\mu}{\sqrt{n}\Delta_x}) - \text{erf}(\frac{a - n\mu}{\sqrt{n}\Delta_x}) \tag{7.50}$$

7.6 Option pricing

A European call option is a contract that depends on an asset S. The contract gives the buyer of the contract the right (the option) to buy S at a fixed price A some time in the future, even if the actual price S may be different. The actual current price of the asset is called the *spot price*. The buyer of the option hopes that the price of the asset, S_t, will exceed A, so that he will be able to buy it at a discount, sell it at market price, and make a profit. The seller of the option hopes this does not happen, so he earns the full sale price. For the buyer of the option, the worst case scenario is not to be able to recover the price paid for the option, but there is no best case because, hypothetically, he can make an arbitrarily large profit. For the seller, it is the opposite. He has an unlimited liability.

In practice, a call option allows a buyer to sell risk (the risk of the price of S going up) to the seller. He pays a price for it, the cost of the option. This is a form of insurance. There are two types of people who trade options: those who are willing to pay to get rid of risk (because they need the underlying asset and want it at a guaranteed price) and those who simply speculate (buy risk and sell insurance). On average, speculators make money because, if they sell many options, risk averages out, and they collect the premiums (the cost of the options).

The European option has a term or expiration, τ. It can only be exercised

at expiration. The amount A is called the *strike price*.

The value at expiration of a European call option is

$$\max(S_\tau - A, 0) \tag{7.51}$$

Its present value is therefore

$$\max(S_\tau - A, 0)e^{-r\tau} \tag{7.52}$$

where r is the risk-free interest rate. This value corresponds to how much we would have to borrow today from a bank so that we can repay the bank at time τ with the profit from the option.

All our knowledge about the future spot price $x = S_\tau$ of the underlying asset can be summarized into a probability mass function $p_\tau(x)$. Under the assumption that $p_\tau(x)$ is known to both the buyer and the seller of the option, it has to be that the averaged net present value of the option is zero for any of the two parties to want to enter into the contract. Therefore

$$C_{call} = e^{-r\tau} \int_{-\infty}^{+\infty} \max(x - A, 0) p_\tau(x) dx \tag{7.53}$$

Similarly, we can perform the same computations for a put option. A put option gives the buyer the option to sell the asset on a given day at a fixed price. This is an insurance against the price going down instead of going up. The value of this option at expiration is

$$\max(A - S_\tau, 0) \tag{7.54}$$

and its pricing formula is

$$C_{put} = e^{-r\tau} \int_{-\infty}^{+\infty} \max(A - x, 0) p_\tau(x) dx \tag{7.55}$$

Also notice that $C_{call} - C_{put} = S_0 - Ae^{-r\tau}$. This relation is called the *call-put parity*.

Our goal is to model $p_\tau(x)$, the distribution of possible prices for the underlying asset at expiration of the option, and compute the preceding integrals using Monte Carlo.

7.6.1 Pricing European options: Binomial tree

To price an option, we need to know $p_\tau(S_\tau)$. This means we need to know something about the future behavior of the price S_τ of the underlying asset S (a stock, an index, or something else). In absence of other information (crystal ball or illegal insider's information), one may try to gather information from a statistical analysis of the past historic data combined with a model of how the price S_τ evolves as a function of time. The most typical model is the binomial model, which is a Wiener process. We assume that the time evolution of the price of the asset X is a stochastic process similar to a random walk. We divide time into intervals of size Δ_t, and we assume that in each time interval $\tau = n\Delta_t$, the variation in the asset price is

$$S_{n+1} = S_n u \text{ with probability } p \tag{7.56}$$

$$S_{n+1} = S_n d \text{ with probability } 1 - p \tag{7.57}$$

where $u > 1$ and $0 < d < 1$ are measures for historic data. It follows that for $\tau = n\Delta_t$, the probability that the spot price of the asset at expiration is $S_u u^i d^{n-i}$ is given by

$$\text{Prob}(S_\tau = S_u u^i d^{n-i}) = \binom{n}{i} p^i (1-p)^{n-i} \tag{7.58}$$

and therefore

$$C_{call} = e^{-r\tau} \frac{1}{n} \sum_{i=0}^{i \le n} \binom{n}{i} p^i (1-p)^{n-i} \max(S_u u^i d^{n-i} - A, 0) \tag{7.59}$$

and

$$C_{put} = e^{-r\tau} \frac{1}{n} \sum_{i=0}^{i \le n} \binom{n}{i} p^i (1-p)^{n-i} \max(A - S_u u^i d^{n-i}, 0) \tag{7.60}$$

The parameters of this model are u, d and p, and they must be determined from historical data. For example,

$$p = \frac{e^{r\Delta_t} - d}{u - d} \tag{7.61}$$

$$u = e^{\sigma\sqrt{\Delta_t}} \tag{7.62}$$

$$d = e^{-\sigma\sqrt{\Delta_t}} \tag{7.63}$$

where Δ_t is the length of the time interval, r is the risk-free rate, and σ is the volatility of the asset, that is, the standard deviation of the log returns.

Here is a Python code to simulate an asset price using a binomial tree:

```
def BinomialSimulation(S0,u,d,p,n):
    data=[]
    S=S0
    for i in xrange(n):
        data.append(S)
        if uniform()<p:
            S=u*S
        else:
            S=d*S
    return data
```

The function takes the present spot value, S_0, of the asset, the values of u, d and p, and the number of simulation steps and returns a list containing the simulated evolution of the stock price. Note that because of the exact formulas, eqs.(7.59) and (7.60), one does not need to perform a simulation unless the underlying asset is a stock that pays dividends or we want to include some other variable in the model.

This method works fine for European call options, but the method is not easy to generalize to other options, when its depends on the path of the asset (e.g., the asset is a stock that pays dividends). Moreover, to increase precision, one has to decrease Δ_t or redo the computation from the beginning.

The Monte Carlo method that we see next is slower in the simple cases but is more general and therefore more powerful.

7.6.2 Pricing European options: Monte Carlo

Here we adopt the Black–Scholes model assumptions. We assume that the time evolution of the price of the asset X is a stochastic process similar to a random walk [62]. We divide time into intervals of size Δ_t, and we assume that in each time interval $t = n\Delta_t$, the log return is a Gaussian random variable:

$$\log \frac{S_{n+1}}{S_n} = \text{gauss}(\mu\Delta_t, \sigma\sqrt{\Delta_t}) \tag{7.64}$$

There are three parameters in the preceding equation:

- Δ_t is the time step we use in our discretization. Δ_t is not a physical parameter; it has nothing to do with the asset. It has to do with the precision of our computation. Let's assume that $\Delta_t = 1$ day.
- μ is a drift term, and it represents the expected rate of return of the asset over a time scale of one year. It is usually set equal to the risk-free rate.
- σ is called volatility, and it represents the number of stochastic fluctuations of the asset over a time interval of one year.

Notice that this model is equivalent to the previous binomial model for large time intervals, in the same sense as the binomial distribution for large values of n approximates the Gaussian distribution. For large T, converge to the same result.

Notice how our assumption that log-return is Gaussian is different and not compatible with Markowitz's assumption of modern portfolio theory (the arithmetic return is Gaussian). In fact, log returns and arithmetic returns cannot both be Gaussian. It is therefore incorrect to optimize a portfolio using MPT when the portfolio includes options priced using Black–Scholes. The price of an individual asset cannot be negative, therefore its arithmetic return cannot be negative and it cannot be Gaussian. Conversely, a portfolio that includes both short and long positions (the holder is the buyer and seller of options) can have negative value. A change of sign in a portfolio is not compatible with the Gaussian log-

return assumption.

If we are pricing a European call option, we are only interested in S_T and not in S_t for $0 < t < T$; therefore we can choose $\Delta_t = T$. In this case, we obtain

$$S_T = S_0 exp(r_T) \tag{7.65}$$

and

$$p(r_T) \propto \exp\left(-\frac{(r_T - \mu T)^2}{2\sigma^2 T}\right) \tag{7.66}$$

This allows us to write the following:

Listing 7.8: in file: `options.py`

```
from nlib import *

class EuropeanCallOptionPricer(MCEngine):
    def simulate_once(self):
        T = self.time_to_expiration
        S = self.spot_price
        R_T = random.gauss(self.mu*T, self.sigma*sqrt(T))
        S_T = S*exp(r_T)
        payoff = max(S_T-self.strike,0)
        return self.present_value(payoff)

    def present_value(self,payoff):
        daily_return = self.risk_free_rate/250
        return payoff*exp(-daily_return*self.time_to_expiration)

def main():
    pricer = EuropeanCallOptionPricer()
    # parameters of the underlying
    pricer.spot_price = 100 # dollars
    pricer.mu = 0.12/250 # daily drift term
    pricer.sigma = 0.30/sqrt(250) # daily variance
    # parameters of the option
    pricer.strike = 110 # dollars
    pricer.time_to_expiration = 90 # days
    # parameters of the market
    pricer.risk_free_rate = 0.05 # 5% annual return

    result = pricer.simulate_many(ap=0.01,rp=0.01) # precision: 1c or 1%
```

```
29      print result
30
31  main()
```

7.6.3 Pricing any option with Monte Carlo

An option is a contract, and one can write a contract with many different clauses. Each of them can be implemented into an algorithm. Yet we can group them into three different categories:

- Non-path-dependent: They depend on the price of the underlying asset at expiration but not on the intermediate prices of the asset (path).
- Weakly path-dependent: They depend on the price of the underlying asset and events that may happen to the price before expiration, but they do not depend on when the events exactly happen.
- Strongly path-dependent: They depend on the details of the time variation of price of the underlying asset before expiration.

Because non-path-dependent options do not depend on details, it is often possible to find approximate analytical formulas for pricing the option. For weakly path-dependent options, usually the binomial tree approach of the previous section is a preferable approach. The Monte Carlo approach applies to the general case, for example, that of strongly path-dependent options.

We will use our `MCEngine` to implement a generic option pricer.

First we need to recognize the following:

- The value of an option at expiration is defined by a payoff function $f(x)$ of the spot price of the asset at the expiration date. The fact that a call option has payoff $f(x) = \max(x - A, 0)$ is a convention that defined the European call option. A different type of option will have a different payoff function $f(x)$.
- The more accurately we model the underlying asset, the more accurate will be the computed value of the option. Some options are more sensitive than others to our modeling details.

Note one never model the option. One only model the underlying asset. The option payoff is given. We only choose the most efficient algorithm based on the model and the option:

Listing 7.9: in file: `options.py`

```
from nlib import *

class GenericOptionPricer(MCEngine):
    def simulate_once(self):
        S = self.spot_price
        path = [S]
        for t in range(self.time_to_expiration):
            r = self.model(dt=1.0)
            S = S*exp(r)
            path.append(S)
        return self.present_value(self.payoff(path))

    def model(self,dt=1.0):
        return random.gauss(self.mu*dt, self.sigma*sqrt(dt))

    def present_value(self,payoff):
        daily_return = self.risk_free_rate/250
        return payoff*exp(-daily_return*self.time_to_expiration)

    def payoff_european_call(self, path):
        return max(path[-1]-self.strike,0)
    def payoff_european_put(self, path):
        return max(path[-1]-self.strike,0)
    def payoff_exotic_call(self, path):
        last_5_days = path[-5]
        mean_last_5_days = sum(last_5_days)/len(last_5_days)
        return max(mean_last_5_days-self.strike,0)

def main():
    pricer = GenericOptionPricer()
    # parameters of the underlying
    pricer.spot_price = 100 # dollars
    pricer.mu = 0.12/250 # daily drift term
    pricer.sigma = 0.30/sqrt(250) # daily variance
    # parameters of the option
    pricer.strike = 110 # dollars
    pricer.time_to_expiration = 90 # days
    pricer.payoff = pricer.payoff_european_call
    # parameters of the market
    pricer.risk_free_rate = 0.05 # 5% annual return

    result = pricer.simulate_many(ap=0.01,rp=0.01) # precision: 1c or 1%
```

```
43      print result
44
45  main()
```

This code allows us to price any option simply by changing the payoff function.

One can also change the model for the underlying using different assumptions. For example, a possible choice is that of including a model for market crashes, and on random days, separated by intervals given by the exponential distribution, assume a negative jump that follows the Pareto distribution (similar to the losses in our previous risk model). Of course, a change of the model requires a recalibration of the parameters.

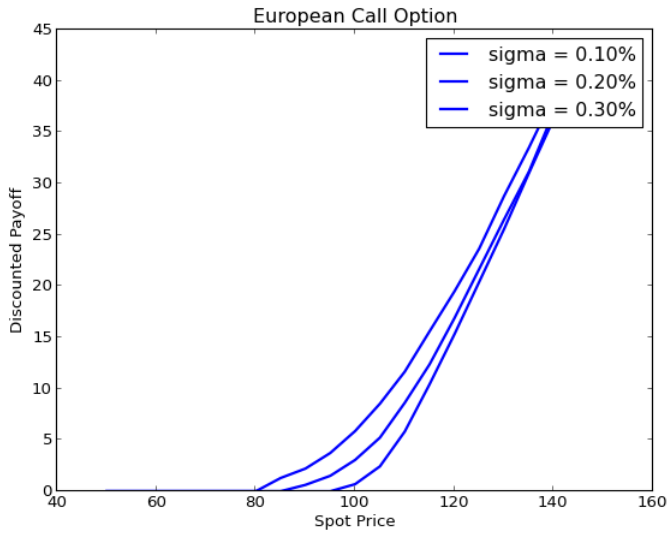

Figure 7.5: Price for a European call option for different spot prices and different values of σ.

7.7 Markov chain Monte Carlo (MCMC) and Metropolis

Until this point, all-out simulations were based on independent random variables. This means that we were able to generate each random num-

ber independently of the others because all the random variables were uncorrelated. There are cases when we have the following problem.

We have to generate $\mathbf{x} = x_0, x_1, \ldots, x_{n-1}$, where $x_0, x_1, \ldots, x_{n-1}$ are n correlated random variables whose probability mass function

$$p(\mathbf{x}) = p(x_0, x_1, \ldots, x_{n-1}) \tag{7.67}$$

cannot be factored, as in $p(x_0, x_1, \ldots, x_{n-1}) = p(x_0)p(x_1)\ldots p(x_{n-1})$. Consider for example the simple case of generating two random numbers x_0 and x_1 both in $[0, 1]$ with probability mass function $p(x_0, x_1) = 6(x_0 - x_1)^2$ (note that $\int_0^1 \int_0^1 6p(x_0, x_1)dx_0dx_1 = 1$, as it should be).

In the case where each of the x_i has a Gaussian distribution and the only dependence between x_i and x_j is their correlation, the solution was already examined in a previous section about the Cholesky algorithm. Here we examine the most general case.

The Metropolis algorithm provides a general and simpler solution to this problem. It is not always the most efficient, but more sophisticated algorithms are nothing but refinements and extensions of its simple idea.

Let's formulate the problem once more: we want to generate $\mathbf{x} = x_0, x_1, \ldots, x_{n-1}$ where $x_0, x_1, \ldots, x_{n-1}$ are n correlated random variables whose probability mass function is given by

$$p(\mathbf{x}) = p(x_0, x_1, \ldots, x_{n-1}) \tag{7.68}$$

The procedure works as follows:

1 Start with a set of independent random numbers $\mathbf{x}^{(0)} = (x_0^{(0)}, x_1^{(0)}, \ldots, x_{n-1}^{(0)})$ in the domain.

2 Generate another set of independent random numbers $\mathbf{x}^{(i+1)} = (x_0^{(i+1)}, x_1^{(i+1)}, \ldots, x_{n-1}^{(i+1)})$ in the domain. This can be done by an arbitrary random function $Q(\mathbf{x}^{(i)})$. The only requirement for this function Q is that the probability of moving from a current point x to a new point y be the same as that of moving from a current point y to a new point x.

3 Generate a uniform random number z.

4 If $p(\mathbf{x}^{(i+1)})/p(\mathbf{x}^{(i)}) < z$, then $\mathbf{x}^{(i+1)} = \mathbf{x}^{(i)}$.

5 Go back to step 2.

The set of random numbers $\mathbf{x}^{(i)}$ generated in this way for large values of i will have a probability mass function given by $p(\mathbf{x})$.

Here is a possible implementation in Python:

```
def metropolis(p,q,x=None):
    while True:
        x_old=x
        x = q(x)
        if p(x)/p(x_old)<random.random()
            x=x_old
        yield x

def P(x):
    return 6.0*(x[0]-x[1])**2

def Q(x):
    return [random.random(), random.random()]

for i, x in enumerate(metropolis(P,Q)):
    print x
    if i==100: break
```

In this example, `Q` is the function that generates random points in the domain (in the example, $[0,1] \times [0,1]$), and `P` is an example probability $p(x) = 6(x_0 - x_1)^2$. Notice we used the Python `yield` function instead of `return`. This means the function is a generator and we can loop over its returned (yielded) values without having to generate all of them at once. They are generated as needed.

Notice that the Metropolis algorithm can generate (and will generate) repeated values. This is because the next random vector x is highly correlated with the previous vector. For this reason, it is often necessary to de-correlate metropolis values by skipping some of them:

```
def metropolis_decorrelate(p,q,x=None,ds=100):
    k = 0
    for x in metropolis(p,q,x):
        k += 1
        if k % ds == ds-1:
             yield x
```

The value of `ds` must be fixed empirically. The value of `ds` which is large enough to make the next vector independent from the previous one is called *decorrelation length*. This generator works as the previous one. For example:

```
for i, x in enumerate(metropolis_decorrelate(P,Q)):
    print x
    if i==100: break
```

7.7.1 The Ising model

A typical example of application of the Metropolis is the Ising model. This model describes a spin system, for example, a ferromagnet. A spin system consists of a regular crystalline structure, and each vertex is an atom. Each atom is a small magnet, and its magnetic orientation can be $+1$ or -1. Each atom interacts with the external magnetic field and with the magnetic field of its six neighbors (think about the six faces of a cube). We use the index i to label an atom and s_i its spin.

The entire system has a total energy given by

$$E(s) = -\sum_i s_i h - \sum_{ij|dist_{ij}=1} s_i s_j \tag{7.69}$$

where h is the external magnetic field, the first sum is over all spin sites, and the second is about all couples of next neighbor sites. In the absence of spin-spin interaction, only the first term contributes, and the energy is lower when the direction of the s_i (their sign) is the same as h. In absence of h, only the second term contributes. The contribution to each couple of spins is positive if their sign is the opposite, and negative otherwise.

In the absence of external forces, each system evolves toward the state of minimum energy, and therefore, for a spin system, each spin tends to align itself in the same direction as its neighbors and in the same direction as the external field h.

Things change when we turn on heat. Feeding energy to the system makes the atoms vibrate and the spins randomly flip. The higher the

temperature, the more they randomly flip.

The probability of finding the system in a given state s at a given temperature T is given by the Boltzmann distribution:

$$p(s) = \exp\left(-\frac{E(s)}{KT}\right) \tag{7.70}$$

where K is the Boltzmann constant.

We can now use the Metropolis algorithm to generate possible states of the system s compatible with a given temperature T and measure the effects on the average magnetization (the average spin) as a function of T and possibly an external field h.

Also notice that in the case of the Boltzmann distribution,

$$\frac{p(s')}{p(s)} = exp\left(\frac{E(s) - E(s')}{KT}\right) \tag{7.71}$$

only depends on the change in energy. The Metropolis algorithm gives us the freedom to choose a function `Q` that changes the state of the system and depends on the current state. We can choose such a function so that we only try to flip one spin at a time. In this case, the `P` algorithm simplifies because we no longer need to compute the total energy of the system at each iteration, but only the variation of energy due to the flipping of that one spin.

Here is the code for a three-dimensional spin system:

Listing 7.10: in file: `ising.py`

```
import random
import math
from nlib import Canvas, mean, sd

class Ising:
    def __init__(self, n):
        self.n = n
        self.s = [[[1 for x in xrange(n)] for y in xrange(n)]
                  for z in xrange(n)]
        self.magnetization = n**3

```

```
    def __getitem__(self,point):
        n = self.n
        x,y,z = point
        return self.s[(x+n)%n][(y+n)%n][(z+n)%n]

    def __setitem__(self,point,value):
        n = self.n
        x,y,z = point
        self.s[(x+n)%n][(y+n)%n][(z+n)%n] = value

    def step(self,t,h):
        n = self.n
        x,y,z = random.randint(0,n-1),random.randint(0,n-1),random.randint(0,n
            -1)
        neighbors = [(x-1,y,z),(x+1,y,z),(x,y-1,z),(x,y+1,z),(x,y,z-1),(x,y,z+1)
            ]
        dE = -2.0*self[x,y,z]*(h+sum(self[xn,yn,zn] for xn,yn,zn in neighbors))
        if dE > t*math.log(random.random()):
            self[x,y,z] = -self[x,y,z]
            self.magnetization += 2*self[x,y,z]
        return self.magnetization

def simulate(steps=100):
    ising = Ising(n=10)
    data = {}
    for h in range(0,11): # external magnetic field
        data[h] = []
        for t in range(1,11): # temperature, in units of K
            m = [ising.step(t=t,h=h) for k in range(steps)]
            mu = mean(m) # average magnetization
            sigma = sd(m)
            data[h].append((t,mu,sigma))
    return data

def main(name='ising.png'):
    data = simulate(steps = 10000)
    canvas = Canvas(xlab='temperature', ylab='magnetization')
    for h in data:
        color = '#%.2x0000' % (h*25)
        canvas.errorbar(data[h]).plot(data[h],color=color)
    canvas.save(name)

main()
```

Fig. 7.7.1 shows how the spins tend to align in the direction of the external magnetic field, but the larger the temperature (left to right), the more random they are, and the average magnetization tends to zero. The higher the external magnetic field (bottom to top curves), the longer it takes for

the transition from order (aligned spins) to chaos (random spins).

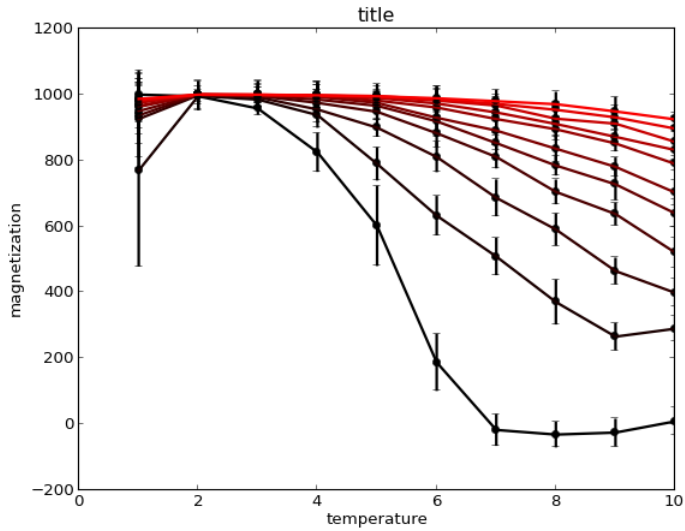

Figure 7.6: Average magnetization as a function of the temperature for a spin system.

Fig. 7.7.1 shows the two-dimensional section of some random three-dimensional states for different values of the temperature. One can clearly see that the lower the temperature, the more the spins are aligned, and the higher the temperature, the more random they are.

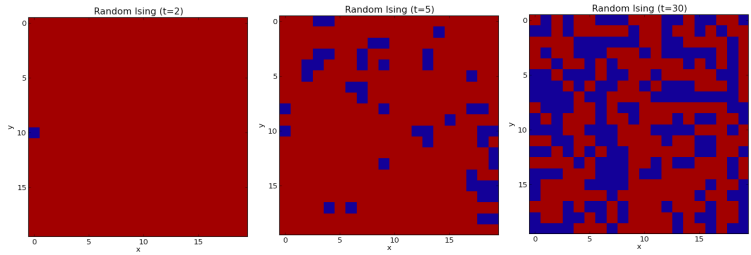

Figure 7.7: Random Ising states (2D section of 3D) for different temperatures.

7.8 Simulated annealing

Simulated annealing is an application of Monte Carlo to solve optimization problems. It is best understood within the context of the Ising model. When the temperature is lowered, the system tends toward the state of minimum energy. At high temperature, the system fluctuates randomly and moves in the space of all possible states. This behavior is not specific to the Ising model. Hence, for any system for which we can define an energy, we can find its minimum energy state, by starting in a random state and slowly lowering the temperature as we evolve the simulation. The system will find a minimum. There may be more than one minimum, and one may need to repeat the procedure multiple times from different initial random states and compare the solutions. This process takes the name of annealing in analogy with the industrial process for removing impurities from metals: heat, cool slowly, repeat.

We can apply this process to any system for which we want to minimize a function $f(x)$ of multiple variables. We just have to think of x as the state s and of f as the energy E. This analogy is purely semantic because the quantity we want to minimize is not necessarily an energy in the physical sense.

Simulated annealing does not assume the function is differentiable or continuous in its variables.

7.8.1 Protein folding

In the following we apply simulated annealing to the problem of folding of a protein. A protein is a sequence of amino-acids. It is normally unfolded, and amino-acids are on a line. When placed in water, it folds. This is because some amino-acids are hydrophobic (repel water) and some are hydrophilic (like contact with water), therefore the protein tries to acquire a three-dimensional shape that minimizes the surface of hydrophobic amino-acids in contact with water [63]. This is represented graphically in fig. 7.8.1.

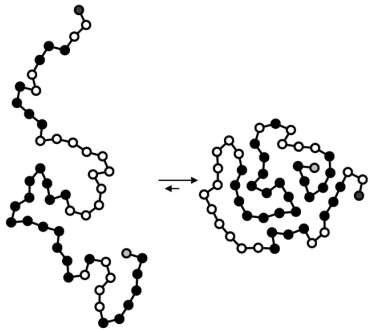

Figure 7.8: Schematic example of protein folding. The white circles are hydrophilic amino-acids. The black ones are hydrophobic.

Here we assume only two types of amino-acids (H for hydrophobic and P for hydrophilic), and we assume each amino acid is a cube, that all cubes have the same size, and that each two consecutive amino-acids are connected at a face. These assumptions greatly simplify the problem because they limit the possible solid angles to six possible values (0: up, 1: down, 2: right, 3: left, 4: front, 5: back). Our goal is arranging the cubes to minimize the number of faces of hydrophobic cubes that are exposed to water:

Listing 7.11: in file: `folding.py`

```
import random
import math
import copy
from nlib import *

class Protein:

    moves = {0:(lambda x,y,z: (x+1,y,z)),
             1:(lambda x,y,z: (x-1,y,z)),
             2:(lambda x,y,z: (x,y+1,z)),
             3:(lambda x,y,z: (x,y-1,z)),
             4:(lambda x,y,z: (x,y,z+1)),
             5:(lambda x,y,z: (x,y,z-1))}

    def __init__(self, aminoacids):
        self.aminoacids = aminoacids
        self.angles = [0]*(len(aminoacids)-1)
        self.folding = self.compute_folding(self.angles)
```

```
        self.energy = self.compute_energy(self.folding)

    def compute_folding(self,angles):
        folding = {}
        x,y,z = 0,0,0
        k = 0
        folding[x,y,z] = self.aminoacids[k]
        for angle in angles:
            k += 1
            xn,yn,zn = self.moves[angle](x,y,z)
            if (xn,yn,zn) in folding: return None # impossible folding
            folding[xn,yn,zn] = self.aminoacids[k]
            x,y,z = xn,yn,zn
        return folding

    def compute_energy(self,folding):
        E = 0
        for x,y,z in folding:
            aminoacid = folding[x,y,z]
            if aminoacid == 'H':
                for face in range(6):
                    if not self.moves[face](x,y,z) in folding:
                        E = E + 1
        return E

    def fold(self,t):
        while True:
            new_angles = copy.copy(self.angles)
            n = random.randint(1,len(self.aminoacids)-2)
            new_angles[n] = random.randint(0,5)
            new_folding = self.compute_folding(new_angles)
            if new_folding: break # found a valid folding
        new_energy = self.compute_energy(new_folding)
        if (self.energy-new_energy) > t*math.log(random.random()):
            self.angles = new_angles
            self.folding = new_folding
            self.energy = new_energy
        return self.energy

def main():
    aminoacids = ''.join(random.choice('HP') for k in range(20))
    protein = Protein(aminoacids)
    t = 10.0
    while t>1e-5:
        protein.fold(t = t)
        print protein.energy, protein.angles
        t = t*0.99 # cool

main()
```

The `moves` dictionary is a dictionary of functions. For each solid angle (0–5), `moves[angle]` is a function that maps `x,y,z`, the coordinates of an amino acid, to the coordinates of the cube at that solid angle.

The annealing procedure is performed in the `main` function. The `fold` procedure is the same step as the Metropolis step. The purpose of the `while` loop in the `fold` function is to find a valid fold for the accept–reject step. Some folds are invalid because they are not physical and would require two amino-acids to occupy the same portion of space. When this happens, the `compute_folding` method returns `None`, indicating that one must try a different folding.

8

Parallel Algorithms

Consider a program that performs the following computation:

```
y = f(x)
z = g(x)
```

In this example, the function $g(x)$ does not depend on the result of the function $f(x)$, and therefore the two functions could be computed independently and in parallel.

Often large problems can be divided into smaller computational problems, which can be solved concurrently ("in parallel") using different processing units (CPUs, cores). This is called *parallel computing*. Algorithms designed to work in parallel are called *parallel algorithms*.

In this chapter, we will refer to a processing unit as a node and to the code running on a node as a process. A parallel program consists of many processes running on as many nodes. It is possible for multiple processes to run on one and the same computing unit (node) because of the multitasking capabilities of modern CPUs, but that is not true parallel computing. We will use an emulator, `Psim`, which does exactly that.

Programs can be parallelized at many levels: bit level, instruction level, data, and task parallelism. Bit-level parallelism is usually implemented in hardware. Instruction-level parallelism is also implemented in hardware in modern multi-pipeline CPUs. Data parallelism is usually referred to as

SIMD. Task parallelism is also referred to as MIMD.

Historically, parallelism was found in applications in high-performance computing, but today it is employed in many devices, including common cell phones. The reason is heat dissipation. It is getting harder and harder to improve speed by increasing CPU frequency because there is a physical limit to how much we can cool the CPU. So the recent trend is keeping frequency constant and increasing the number of processing units on the same chip.

Parallel architectures are classified according to the level at which the hardware supports parallelism, with multicore and multiprocessor computers having multiple processing elements within a single machine, while clusters, MPPs, and grids use multiple computers to work on the same task. Specialized parallel computer architectures are sometimes used alongside traditional processors for accelerating specific tasks.

Optimizing an algorithm to run on a parallel architecture is not an easy task. Details depend on the type of parallelism and details of the architecture.

In this chapter, we will learn how to classify architectures, compute running times of parallel algorithms, and measure their performance and scaling.

We will learn how to write parallel programs using standard programming patterns, and we will use them as building blocks for more complex algorithms.

For some parts of this chapter, we will use a simulator called `PSim`, which is written in Python. Its performances will only scale on multicore machines, but it will allow us to emulate various network topologies.

8.1 Parallel architectures

8.1.1 Flynn taxonomy

Parallel computer architecture classifications are known as Flynn's taxonomy [64] and are due to the work of Michael J. Flynn in 1966.

Flynn identified the following architectures:

- **Single instruction, single data stream (SISD)**

  A sequential computer that exploits no parallelism in either the instruction or data streams. A single control unit (CU) fetches a single instruction stream (IS) from memory. The CU then generates appropriate control signals to direct single processing elements (PE) to operate on a single data stream (DS), for example, one operation at a time.

  Examples of SISD architecture are the traditional uniprocessor machines like a PC (currently manufactured PCs have multiple processors) or old mainframes.

- **Single instruction, multiple data streams (SIMD)** A computer that exploits multiple data streams against a single instruction stream to perform operations that may be naturally parallelized (e.g., an array processor or GPU).

- **Multiple instruction, single data stream (MISD)**

  Multiple instructions operate on a single data stream. This is an uncommon architecture that is generally used for fault tolerance. Heterogeneous systems operate on the same data stream and must agree on the result. Examples include the now retired Space Shuttle flight control computer.

- **Multiple instruction, multiple data streams (MIMD)** Multiple autonomous processors simultaneously executing different instructions on different data. Distributed systems are generally recognized to be MIMD architectures, either exploiting a single shared memory space (using threads) or a distributed memory space (using a message-passing protocol such as MPI).

MIMD can be further subdivided into the following:

- **Single program, multiple data (SPMD)** Multiple autonomous processors simultaneously executing the same program but at independent points not synchronously (as in the SIMD case). SPMD is the most common style of parallel programming.
- **Multiple program, multiple data (MPMD)** Multiple autonomous processors simultaneously operating at least two independent programs. Typically such systems pick one node to be the "host" ("the explicit host/node programming model") or "manager" (the "manager–worker" strategy), which runs one program that farms out data to all the other nodes, which all run a second program. Those other nodes then return their results directly to the manager. The Map-Reduce pattern also falls under this category.

An embarrassingly parallel workload (or embarrassingly parallel problem) is one for which little or no effort is required to separate the problem into a number of parallel tasks. This is often the case where there exists no dependency (or communication) between those parallel tasks.

The manager–worker node strategy, when workers do not need to communicate with each other, is an example of an "embarrassingly parallel" problem.

8.1.2 Network topologies

In the MIMD case, multiple copies of the same problem run concurrently (on different data subsets and branching differently, thus performing different instructions) on different processing units, and they exchange information using a network. How fast they can communicate depends on the network characteristics identified by the network topology and the latency and bandwidth of the individual links of the network.

Normally we classify network topologies based on the following taxonomy:

- **Completely connected:** Each node is connected by a directed link to each other node.

- **Bus topology:** All nodes are connected to the same single cable. Each computer can therefore communicate with each other computer using one and the same bus cable. The limitation of this approach is that the communication bandwidth is limited by the bandwidth of the cable. Most bus networks only allow two machines to communicate with each other at one time (with the exception of one too many broadcast messages). While two machines communicate, the others are stuck waiting. The bus topology is the most inexpensive but also slow and constitutes a single point of failure.
- **Switch topology (star topology):** In local area networks with a switch topology, each computer is connected via a direct link to a central device, usually a switch, and it resembles a star. Two computers can communicate using two links (to the switch and from the switch). The central point of failure is the switch. The switch is usually intelligent and can reroute the messages from any computer to any other computer. If the switch has sufficient bandwidth, it can allow multiple computers to talk to each other at the same time. For example, for a 10 Gbit/s links and an 80 Gbit/s switch, eight computers can talk to each other (in pairs) at the same time.
- **Mesh topology:** In a mesh topology, computers are assembled into an array (1D, 2D, etc.), and each computer is connected via a direct link to the computers immediately close (left, right, above, below, etc.). Next neighbor communication is very fast because it involves a single link and therefore low latency. For two computers not physically close to communicate, it is necessary to reroute messages. The latency is proportional to the distance in links between the computers. Some meshes do not support this kind of rerouting because the extra logic, even if unused, may be cause for extra latency. Meshes are ideal for solving numerical problems such as solving differential equations because they can be naturally mapped into this kind of topology.
- **Torus topology:** Very similar to a mesh topology (1D, 2D, 3D, etc.), except that the network wraps around the edges. For example, in one dimension node, i is connected to $(i+1)\%p$, where p is the total number of nodes. A one-dimensional torus is called a *ring network*.

- **Tree network:** The tree topology looks like a tree where the computer may be associated with every tree node or every leaf only. The tree links are the communication link. For a binary tree, each computer only talks to its parent and its two children nodes. The root node is special because it has no parent node.

  Tree networks are ideal for global operations such as broadcasting and for sharing IO devices such as disks. If the IO device is connected to the root node, every other computer can communicate with it using only $\log p$ links (where p is the number of computers connected). Moreover, each subset of a tree network is also a tree network. This makes it easy to distribute subtasks to different subsets of the same architecture.

- **Hypercube:** This network assumes 2^d nodes, and each node corresponds to a vertex of a hypercube. Nodes are connected by direct links, which correspond to the edges of the hypercube. Its importance is more academic than practical, although some ideas from hypercube networks are implemented in some algorithms.

If we identify each node on the network with a unique integer number called its rank, we write explicit code to determine if two nodes i and j are connected for each network topology:

Listing 8.1: in file: `psim.py`

```
import os, string, pickle, time, math

def BUS(i,j):
    return True

def SWITCH(i,j):
    return True

def MESH1(p):
    return lambda i,j,p=p: (i-j)**2==1

def TORUS1(p):
    return lambda i,j,p=p: (i-j+p)%p==1 or (j-i+p)%p==1

def MESH2(p):
    q=int(math.sqrt(p)+0.1)
    return lambda i,j,q=q: ((i%q-j%q)**2,(i/q-j/q)**2) in [(1,0),(0,1)]

```

```
def TORUS2(p):
    q=int(math.sqrt(p)+0.1)
    return lambda i,j,q=q: ((i%q-j%q+q)%q,(i/q-j/q+q)%q) in [(0,1),(1,0)] or \
                           ((j%q-i%q+q)%q,(j/q-i/q+q)%q) in [(0,1),(1,0)]
def TREE(i,j):
    return i==int((j-1)/2) or j==int((i-1)/2)
```

8.1.3 Network characteristics

- **Number of links**
- **Diameter:** The max distance between any two nodes measured as a minimum number of links connecting them. Smaller diameter means smaller latency. The diameter is proportional to the maximum time it takes for a message go from one node to another.
- **Bisection width:** The minimum number of links one has to cut to turn the network into two disjoint networks. Higher bisection width means higher reliability of the network.
- **Arc connectivity:** The number of different paths (non-overlapping and of minimal length) connecting any two nodes. Higher connectivity means higher bandwidth and higher reliability.

Here are values of this parameter for each type of network:

Network	Links	Diameter	Width
completely connected	$p(p-1)/2$	1	$p-1$
switch	p	2	1
1D mesh	$p-1$	$p-1$	1
nD mesh	$n(p^{\frac{1}{n}}-1)p^{\frac{n-1}{n}}$	n	$p^{\frac{2}{3}}$
1D torus	p	$\frac{p}{2}$	2
nD torus	np	$\frac{n}{2}p^{\frac{1}{n}}$	$2n$
hypercube	$\frac{p}{2}\log_2 p$	$\log_2 p$	$log_2 p$
tree	$p-1$	$\log_2 p$	1

Most actual supercomputers implement a variety of taxonomies and topologies simultaneously. A modern supercomputer has many nodes, each node has many CPUs, each CPU has many cores, and each core im-

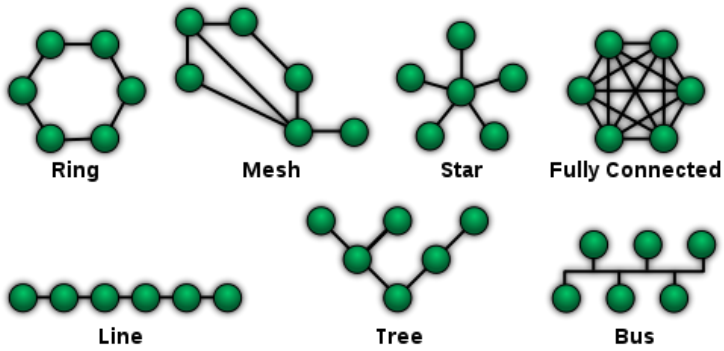

Figure 8.1: Examples of network topologies.

plements SIMD instructions. Each core has it own cache, each CPU has its own cache, and each node has its own memory shared by all threads running on that one node. Nodes communicate with each other using multiple networks (typically a multidimensional mesh for point-to-point communications and a tree network for global communication and general disk IO).

This makes writing parallel programs very difficult. Parallel programs must be optimized for each specific architecture.

8.2 Parallel metrics

8.2.1 Latency and bandwidth

The time it takes for a message of size m (in bytes) over a wire can be broken into two components: a fixed overall time that does not depend on the size of the message, called *latency* (and indicated with t_s), and a time proportional to the message size, called *inverse bandwidth* (and indicated with t_w).

Think of a pipe of length L and section s, and you want to pump m liters of water through the pipe at velocity v. From the moment you start pumping, it takes L/v seconds before the water starts arriving at the other

end of the pipe. From that moment, it will take m/sv for all the water to arrive at its destination. In this analogy, L/v is the latency t_s, sv is the bandwidth, and $t_w = 1/sv$.

The total time to send the message (or the water) is

$$T(m) = t_s + t_w m \tag{8.1}$$

From now on, we will use $T_1(n)$ to refer to the nonparallel running time of an algorithm as a function of its input m. We will use $T_p(n)$ to refer to its running time with p parallel processes.

As a practical case, in the following example, we consider a generic algorithm with the following parallel and nonparallel running times:

$$T_1(n) = t_a n^2 \tag{8.2}$$
$$T_p(n) = t_a n^2/p + 2p(t_s + t_w n/p) \tag{8.3}$$

These formulas may come from example from the problem of multiplying a matrix times a vector.

Here t_a is the time to perform one elementary instruction; t_s and t_w are the latency and inverse bandwidth. The first term of T_p is nothing but T_1/p, while the second term is an overhead due to communications.

Typically $t_s >> t_w >> t_a$. In the following plots, we will always assume $t_a = 1$, $t_s = 0.2$, and $t_w = 0.1$. With these assumptions, fig. 8.2.1 shows how T_p changes with input size and number of parallel processes. Notice that while for small p, T_p decreases $\propto 1/p$, for large p, the communication overhead dominates over computation. This overhead is our example and is dominated by the latency contribution, which grows with p.

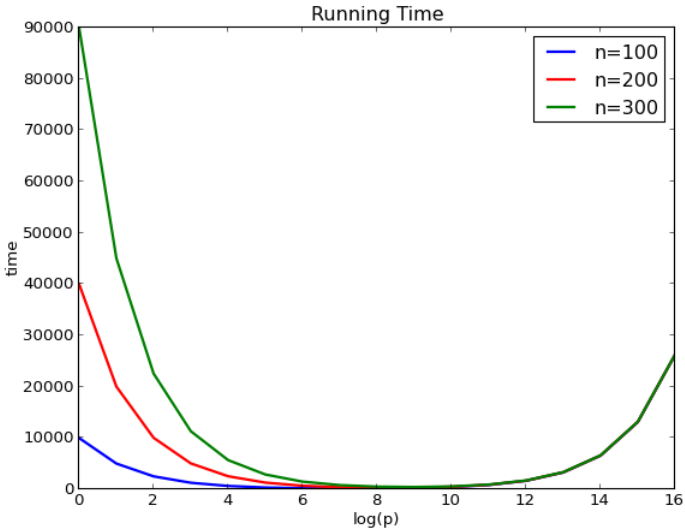

Figure 8.2: T_p as a function of input size n and number of processes p.

8.2.2 Speedup

The *speedup* is defined as

$$S_p(n) = \frac{T_1(n)}{T_p(n)} \tag{8.4}$$

where T_1 is the time it takes to run the algorithm on an input of size n on one processing unit (e.g., node), and T_p is the time it takes to run the same algorithm on the same input using p nodes in parallel. Fig. 8.2.2 shows an example of speedup. When communication overhead dominates, speedup decreases.

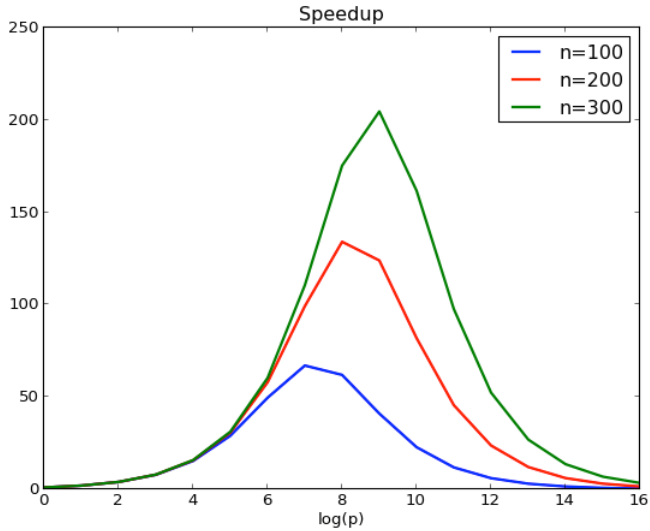

Figure 8.3: S_p as a function of input size n and number of processes p.

8.2.3 Efficiency

The *efficiency* is defined as

$$E_p(n) = \frac{S_p(n)}{p} = \frac{T_1(n)}{pT_p(n)} \tag{8.5}$$

Notice that in case of perfect parallelization (impossible), $T_p = T_1/p$, and therefore $E_p(n) = 1$. Fig. 8.2.3 shows an example of efficiency. When communication overhead dominates, efficiency drops. Notice efficiency is always less than 1. We do not write parallel algorithms because they are more efficient. They are always less efficient and more costly than the nonparallel ones. We do it because we want the result sooner, and there is an economic value in it.

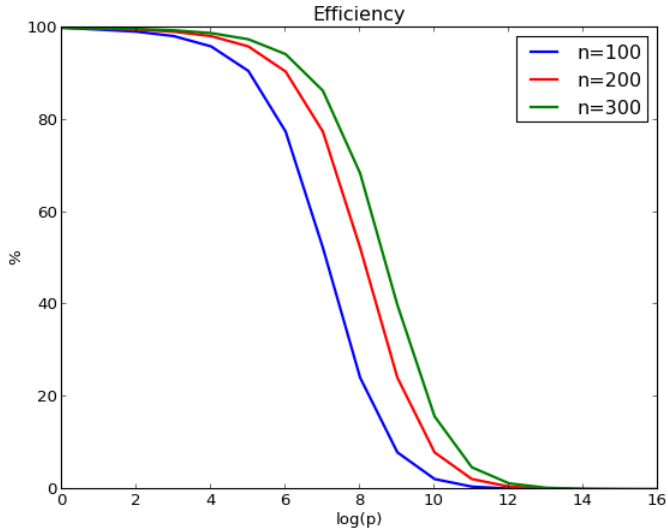

Figure 8.4: E_p as a function of input size n and number of processes p.

8.2.4 Isoefficiency

Given a value of efficiency that we choose as target, E, and a given number of nodes, p, we ask what is the maximum size of a problem that we can solve. The answer is found by solving n the following equation:

$$E_p(n) = E \tag{8.6}$$

For example T_p, we obtain

$$E_p = \frac{1}{1 + 2p^2(t_s + t_w n/p)/(n^2 t_a)} = E \tag{8.7}$$

which solved in n yields

$$n \simeq 2\frac{t_w}{t_a}\frac{E}{1-E}p \tag{8.8}$$

Isoefficiency curves for different values of E are shown in fig. 8.2.4. For our example problem, n is proportional to p. In general, this is not true, but n is monotonic in p.

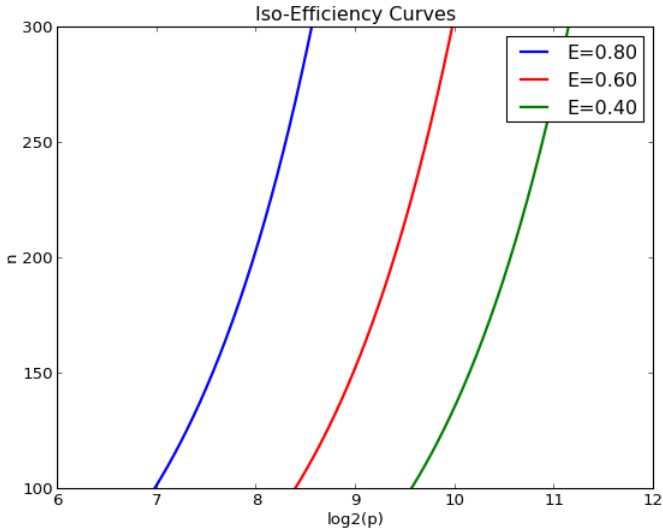

Figure 8.5: Isoefficiency curves for different values of the target efficiency.

8.2.5 Cost

The cost of a computation is equal to the time it takes to run on each node, multiplied by the number of nodes involved in the computation:

$$C_p(n) = pT_p(n) \tag{8.9}$$

Notice that in general

$$\frac{\mathrm{d}C_p(n)}{\mathrm{d}p} = \alpha T_1(n) > 0 \tag{8.10}$$

This means that for a fixed problem size n, the more an algorithm is parallelized, the more it costs to run it (because it gets less and less efficient).

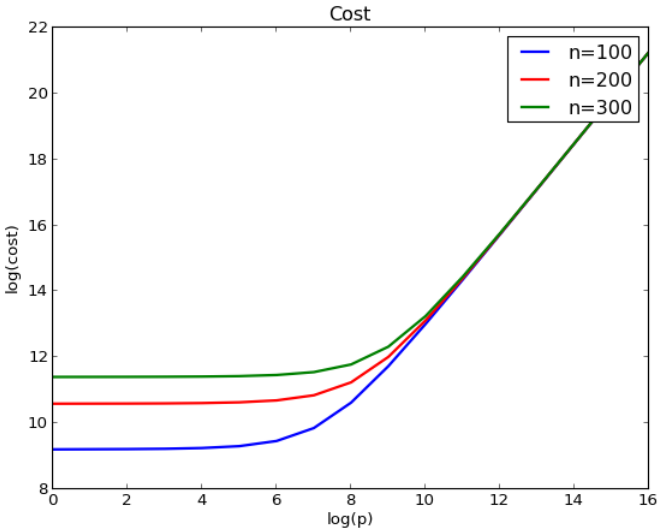

Figure 8.6: C_p as a function of input size n and a number of processes p.

8.2.6 Cost optimality

With the preceding disclaimer, we define cost optimality as the choice of p (as a function of n), which makes the cost scale proportional to $T_1(n)$:

$$pT_p(n) \propto T_1(n) \tag{8.11}$$

Or in other words, looking for the $p(n)$ such that

$$\lim_{n\to\infty} p(n)T_{p(n)}(n)/T_1(n) = const. \neq 0 \tag{8.12}$$

8.2.7 Amdahl's law

Consider an algorithm that can be parallelized, but one faction α of its total sequential running time αT_1 cannot be parallelized. That means that

$T_p = \alpha T_1 + (1-\alpha)T_1/p$, and this yields [65]

$$S_p = \frac{1}{\alpha + (1-\alpha)/p} < \frac{1}{\alpha} \tag{8.13}$$

Therefore the speedup is theoretically limited.

8.3 Message passing

Consider the following Python program:

```
def f():
    import os
    if os.fork(): print True
    else: print False
f()
```

The output of the current program is

```
True
False
```

The function `fork` creates a copy of the current process (a child). The parent process returns the ID of the child process, and the child process returns 0. Therefore the `if` condition is both true and false, just on different processes.

We have created a Python module called `psim`, and its source code is listed here; `psim` forks the parallel processes, creates sockets connecting them, and provides API for communications. An example of `psim` usage will be given later.

Listing 8.2: in file: `psim.py`

```
class PSim(object):
    def log(self,message):
        """
        logs the message into self._logfile
        """
        if self.logfile!=None:
            self.logfile.write(message)

    def __init__(self,p,topology=SWITCH,logfilename=None):
        """
```

```
        forks p-1 processes and creates p*p
        """
        self.logfile = logfilename and open(logfile,'w')
        self.topology = topology
        self.log("START: creating %i parallel processes\n" % p)
        self.nprocs = p
        self.pipes = {}
        for i in xrange(p):
            for j in xrange(p):
                self.pipes[i,j] = os.pipe()
        self.rank = 0
        for i in xrange(1,p):
            if not os.fork():
                self.rank = i
                break
        self.log("START: done.\n")

    def send(self,j,data):
        """
        sends data to process #j
        """
        if not self.topology(self.rank,j):
            raise RuntimeError('topology violation')
        self._send(j,data)

    def _send(self,j,data):
        """
        sends data to process #j ignoring topology
        """
        if j<0 or j>=self.nprocs:
            self.log("process %i: send(%i,...) failed!\n" % (self.rank,j))
            raise Exception
        self.log("process %i: send(%i,%s) starting...\n" % \
                 (self.rank,j,repr(data)))
        s = pickle.dumps(data)
        os.write(self.pipes[self.rank,j][1], string.zfill(str(len(s)),10))
        os.write(self.pipes[self.rank,j][1], s)
        self.log("process %i: send(%i,%s) success.\n" % \
                 (self.rank,j,repr(data)))

    def recv(self,j):
        """
        returns the data received from process #j
        """
        if not self.topology(self.rank,j):
            raise RuntimeError('topology violation')
        return self._recv(j)

    def _recv(self,j):
```

```
        """
        returns the data received from process #j ignoring topology
        """
        if j<0 or j>=self.nprocs:
            self.log("process %i: recv(%i) failed!\n" % (self.rank,j))
            raise RuntimeError
        self.log("process %i: recv(%i) starting...\n" % (self.rank,j))
        try:
            size=int(os.read(self.pipes[j,self.rank][0],10))
            s=os.read(self.pipes[j,self.rank][0],size)
        except Exception, e:
            self.log("process %i: COMMUNICATION ERROR!!!\n" % (self.rank))
            raise e
        data=pickle.loads(s)
        self.log("process %i: recv(%i) done.\n" % (self.rank,j))
        return data
```

An instance of the class `PSim` is an object that can be used to determine the total number of parallel processes, the rank of each running process, to send messages to other processes, and to receive messages from them. It is usually called a *communicator*; `send` and `recv` represent the simplest type of communication pattern, point-to-point communication.

A PSim program starts by importing and creating an instance of the `PSim` class. The constructor takes two arguments, the number of parallel processes you want and the network topology you want to emulate. Before returning the `PSim` instance, the constructor makes $p-1$ copies of the running process and creates sockets connecting each two of them. Here is a simple example in which we make two parallel processes and send a message from process 0 to process 1:

```
from psim import *

comm = PSim(2,SWITCH)
if comm.rank == 0:
    comm.send(1, "Hello World")
elif comm.rank == 1:
    message = comm.recv(0)
    print message
```

Here is a more complex example that creates $p = 10$ parallel processes, and node 0 sends a message to each one of them:

```
from psim import *

p = 10
```

```
4
5  comm = PSim(p,SWITCH)
6  if comm.rank == 0:
7      for other in range(1,p):
8         comm.send(other, "Hello %s" % p)
9  else:
10     message = comm.recv(0)
11     print message
```

Following is a more complex example that implements a parallel scalar product. The process with rank 0 makes up two vectors and distributes pieces of them to the other processes. Each process computes a part of the scalar product. Of course, the scalar product runs in linear time, and it is very inefficient to parallelize it, yet we do it for didactic purposes.

Listing 8.3: in file: `psim_scalar.py`

```
1  import random
2  from psim import PSim
3
4  def scalar_product_test1(n,p):
5      comm = PSim(p)
6      h = n/p
7      if comm.rank==0:
8          a = [random.random() for i in xrange(n)]
9          b = [random.random() for i in xrange(n)]
10         for k in xrange(1,p):
11             comm.send(k, a[k*h:k*h+h])
12             comm.send(k, b[k*h:k*h+h])
13     else:
14         a = comm.recv(0)
15         b = comm.recv(0)
16     scalar = sum(a[i]*b[i] for i in xrange(h))
17     if comm.rank == 0:
18         for k in xrange(1,p):
19             scalar += comm.recv(k)
20         print scalar
21     else:
22         comm.send(0,scalar)
23
24 scalar_product_test(10,2)
```

Most parallel algorithms follow a similar pattern. One process has access to IO. That process reads and scatters the data. The other processes perform their part of the computation; the results are reduced (aggregated) and sent back to the root process. This pattern may be repeated by multiple functions, perhaps in loops. Different functions may handle different

data structure and may have different communication patterns. The one thing that must be constant throughout the run is the number of processes because one wants to pair each process with one computing unit.

In the following, we implement a parallel version of the `mergesort`. At each step, the code splits the problem into two smaller problems. Half of the problem is solved by the process that performed the split and assigns the other half to an existing free process. When there are no more free processes, it reverts to the nonparallel `mergesort` step. The `merge` function here is the same as the nonparallel `mergesort` of chapter 3.

Listing 8.4: in file: `psim_mergesort.py`

```
import random
from psim import PSim

def mergesort(A, x=0, z=None):
    if z is None: z = len(A)
    if x<z-1:
        y = int((x+z)/2)
        mergesort(A,x,y)
        mergesort(A,y,z)
        merge(A,x,y,z)

def merge(A,x,y,z):
    B,i,j = [],x,y
    while True:
        if A[i]<=A[j]:
            B.append(A[i])
            i=i+1
        else:
            B.append(A[j])
            j=j+1
        if i==y:
            while j<z:
                B.append(A[j])
                j=j+1
            break
        if j==z:
            while i<y:
                B.append(A[i])
                i=i+1
            break
    A[x:z]=B

def mergesort_test(n,p):
```

```
    comm = PSim(p)
    if comm.rank==0:
        data = [random.random() for i in xrange(n)]
        comm.send(1, data[n/2:])
        mergesort(data,0,n/2)
        data[n/2:] = comm.recv(1)
        merge(data,0,n/2,n)
        print data
    else:
        data = comm.recv(0)
        mergesort(data)
        comm.send(0,data)

mergesort_test(20,2)
```

More interesting patterns are global communication patterns implemented on top of `send` and `recv`. Subsequently, we discuss the most common: broadcast, scatter, collect, and reduce. Our implementation is not the most efficient, but it is the simplest. In principle, there should be a different implementation for each type of network topology to take advantage of its features.

8.3.1 Broadcast

The simplest type of broadcast is the one-2-all, which consists of one process (source) sending a message (value) to every other process. A more complex broadcast is when each process broadcasts a message simultaneously and each node receives the list of values ordered by the rank of the sender:

Listing 8.5: in file: `psim.py`

```
    def one2all_broadcast(self, source, value):
        self.log("process %i: BEGIN one2all_broadcast(%i,%s)\n" % \
                 (self.rank,source, repr(value)))
        if self.rank==source:
            for i in xrange(0, self.nprocs):
                if i!=source:
                    self._send(i,value)
        else:
            value=self._recv(source)
        self.log("process %i: END one2all_broadcast(%i,%s)\n" % \
                 (self.rank,source, repr(value)))
        return value
```

```
13
14      def all2all_broadcast(self, value):
15          self.log("process %i: BEGIN all2all_broadcast(%s)\n" % \
16                   (self.rank, repr(value)))
17          vector=self.all2one_collect(0,value)
18          vector=self.one2all_broadcast(0,vector)
19          self.log("process %i: END all2all_broadcast(%s)\n" % \
20                   (self.rank, repr(value)))
21          return vector
```

We have implemented the all-to-all broadcast using a trick. We send collected all values to node with rank 0 (via a function collect), and then we did a one-to-all broadcast of the entire list from node 0. In general, the implementation depends on the topology of the available network.

Here is an example of an application of broadcasting:

```
1 from psim import *
2
3 p = 10
4
5 comm = PSim(p,SWITCH)
6 message = "Hello World" if comm.rank==0 else None
7 message = comm.one2all_broadcast(0, message)
8 print message
```

Notice how before the broadcast, only the process with rank 0 has knowledge of the message. After broadcast, all nodes are aware of it. Also notice that `one2all_broadcast` is a global communication function, and all processes must call it. Its first argument is the rank of the broadcasting process (0), while the second argument is the message to be broadcast (only the value from node 0 is actually used).

8.3.2 Scatter and collect

The all-to-one collect pattern works as follows. Every process sends a value to process `destination`, which receives the values in a list ordered according to the rank of the senders:

Listing 8.6: in file: `psim.py`

```
1       def one2all_scatter(self,source,data):
2           self.log('process %i: BEGIN all2one_scatter(%i,%s)\n' % \
3                    (self.rank,source,repr(data)))
```

```
        if self.rank==source:
             h, remainder = divmod(len(data),self.nprocs)
             if remainder: h+=1
             for i in xrange(self.nprocs):
                 self._send(i,data[i*h:i*h+h])
        vector = self._recv(source)
        self.log('process %i: END all2one_scatter(%i,%s)\n' % \
                 (self.rank,source,repr(data)))
        return vector

    def all2one_collect(self,destination,data):
        self.log("process %i: BEGIN all2one_collect(%i,%s)\n" % \
                 (self.rank,destination,repr(data)))
        self._send(destination,data)
        if self.rank==destination:
            vector = [self._recv(i) for i in xrange(self.nprocs)]
        else:
            vector = []
        self.log("process %i: END all2one_collect(%i,%s)\n" % \
                 (self.rank,destination,repr(data)))
        return vector
```

Here is a revised version of the previous scalar product example using scatter:

Listing 8.7: in file: `psim_scalar2.py`

```
import random
from psim import PSim

def scalar_product_test2(n,p):
    comm = PSim(p)
    a = b = None
    if comm.rank==0:
        a = [random.random() for i in xrange(n)]
        b = [random.random() for i in xrange(n)]
    a = comm.one2all_scatter(0,a)
    b = comm.one2all_scatter(0,b)

    scalar = sum(a[i]*b[i] for i in xrange(len(a)))

    scalar = comm.all2one_reduce(0,scalar)
    if comm.rank == 0:
        print scalar

scalar_product_test2(10,2)
```

8.3.3 Reduce

The all-to-one reduce pattern is very similar to the collect, except that the `destination` does not receive the entire list of values but some aggregated information about the values. The aggregation must be performed using a commutative binary function $f(x,y) = f(y,x)$. This guarantees that the reduction from the values go down in any order and thus are optimized for different network topologies.

The all-to-all reduce is similar to reduce, but every process will get the result of the reduction, not just one `destination` node. This may be achieved by an all-to-one reduce followed by a one-to-all broadcast:

Listing 8.8: in file: `psim.py`

```
    def all2one_reduce(self,destination,value,op=lambda a,b:a+b):
        self.log("process %i: BEGIN all2one_reduce(%s)\n" % \
                 (self.rank,repr(value)))
        self._send(destination,value)
        if self.rank==destination:
            result = reduce(op,[self._recv(i) for i in xrange(self.nprocs)])
        else:
            result = None
        self.log("process %i: END all2one_reduce(%s)\n" % \
                 (self.rank,repr(value)))
        return result

    def all2all_reduce(self,value,op=lambda a,b:a+b):
        self.log("process %i: BEGIN all2all_reduce(%s)\n" % \
                 (self.rank,repr(value)))
        result=self.all2one_reduce(0,value,op)
        result=self.one2all_broadcast(0,result)
        self.log("process %i: END all2all_reduce(%s)\n" % \
                 (self.rank,repr(value)))
        return result
```

And here are some examples of a reduce operation that can be passed to the `op` argument of the `all2one_reduce` and `all2all_reduce` methods:

Listing 8.9: in file: `psim.py`

```
        @staticmethod
        def sum(x,y): return x+y
        @staticmethod
        def mul(x,y): return x*y
        @staticmethod
```

```
        def max(x,y): return max(x,y)
        @staticmethod
        def min(x,y): return min(x,y)
```

Graph algorithms can also be parallelized, for example, the Prim algorithm. One way to do it is to represent the graph using an adjacency matrix where term i, j corresponds to the link between vertex i and vertex j. The term can be `None` if the link does not exist. Any graph algorithm, in some order, loops over the vertices and over the neighbors. This step can be parallelized by assigning different columns of the adjacency matrix to different computing processes. Each process only loops over some of the neighbors of the vertex being processed. Here is an example of the Prim algorithm:

Listing 8.10: in file: `psim_prim.py`

```
from psim import PSim
import random

def random_adjacency_matrix(n):
    A = []
    for r in range(n):
        A.append([0]*n)
    for r in range(n):
        for c in range(0,r):
            A[r][c] = A[c][r] = random.randint(1,100)
    return A

class Vertex(object):
    def __init__(self,path=[0,1,2]):
        self.path = path

def weight(path=[0,1,2], adjacency=None):
    return sum(adjacency[path[i-1]][path[i]] for i in range(1,len(path)))

def bb(adjacency,p=1):
    n = len(adjacency)
    comm = PSim(p)
    Q = []
    path = [0]
    Q.append(Vertex(path))
    bound = float('inf')
    optimal = None
    local_vertices = comm.one2all_scatter(0,range(n))
    while True:
        if comm.rank==0:
```

```
            vertex = Q.pop() if Q else None
        else:
            vertex = None
        vertex = comm.one2all_broadcast(0,vertex)
        if vertex is None:
            break
        P = []
        for k in local_vertices:
            if not k in vertex.path:
                new_path = vertex.path+[k]
                new_path_length = weight(new_path,adjacency)
                if new_path_length<bound:
                    if len(new_path)==n:
                        new_path.append(new_path[0])
                        new_path_length = weight(new_path,adjacency)
                        if new_path_length<bound:
                            bound = new_path_length # bcast
                            optimal = new_path      # bcast
                    else:
                        new_vertex = Vertex(new_path)
                        P.append(new_vertex) # fix this
                print new_path, new_path_length
        x = (bound,optimal)
        x = comm.all2all_reduce(x,lambda a,b: min(a,b))
        (bound,optimal) = x

        P = comm.all2one_collect(0,P)
        if comm.rank==0:
            for item in P:
                Q+=item
    return optimal, bound


m = random_adjacency_matrix(5)
print bb(m,p=2)
```

8.3.4 Barrier

Another global communication pattern is the barrier. It forces all processes when they reach the barrier to stop and wait until all the other processes have reached the barrier. Think of runners gathering at the starting line of a race; when all the runners are there, the race can start.

Here we implement it using a simple all-to-all broadcast:

Listing 8.11: in file: psim.py

```
    def barrier(self):
        self.log("process %i: BEGIN barrier()\n" % (self.rank))
        self.all2all_broadcast(0)
        self.log("process %i: END barrier()\n" % (self.rank))
        return
```

The use of barrier is usually a symptom of bad code because it forces parallel processes to wait for other processes without data actually being transferred.

8.3.5 Global running times

In the following table, we compute the order of growth or typical running times for the most common network topologies for typical communication algorithms:

Network	Send/Recv	One2All Bcast	Scatter
completely connected	1	1	1
switch	2	$\log p$	$2p$
1D mesh	$p-1$	$p-1$	p^2
2D mesh	$2(p^{\frac{1}{2}}-1)$	$\sqrt{p}$	p^2
3D mesh	$3(p^{\frac{1}{3}}-1)$	$p^{1/3}$	p^2
1D torus	$p/2$	$p/2$	p^2
2D torus	$2p^{\frac{1}{2}}$	$\sqrt{p}/2$	p^2
3D torus	$3/2p^{\frac{1}{3}}$	$p^{1/3}/2$	p^2
hypercube	$\log p$	$\log_2 p$	p
tree	$\log p$	$\log p$	p

It is obvious that the completely connected is the fastest network but also the most expensive to build. The tree is a cheap compromise. The switch tends to be faster for arbitrary point-to-point communication, but the switch comes to a premium. Multidimensional meshes and toruses become cost-effective when solving problems that are naturally defined on a grid because they only require next neighbor interaction.

8.4 mpi4py

The Psim emulator does not provide any actual speedup unless you have multiple cores or processors to execute the forked processes. A better approach would be to use `mpi4py` [66] because it allows running different processes on different machines on a network. `mpi4py` is a Python interface to the message passing interface (MPI). MPI is a standard protocol and API for interprocess communications. Its API are equivalent one by one to those of `PSim`, except that they have different names and different signatures.

Here is an example of using mpi4py:

```
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

if rank == 0:
   message = "Hello World"
   comm.send(message, dest=1, tag=11)
elif rank == 1:
   message = comm.recv(source=0, tag=11)
   print message
```

The `comm` object of class `MPI.COMM_WORLD` plays a similar role as the `PSim` object of the previous section. The MPI `send` and `recv` functions are very similar to the `PSim` equivalent ones, except that they require details about the type of the data being transferred and a communication tag. The tag allows node A to send multiple messages to B and allows B to receive them out of order. PSim does not allow tags.

8.5 Master-Worker and Map-Reduce

Map-Reduce [67] is a framework for processing highly distributable problems across huge data sets using a large number of computers (nodes). The group of computers is collectively referred to as a cluster (if all nodes use the same hardware) or a grid (if the nodes use different hardware). It comprises two steps:

"Map" (implemented here in a function `mapfn`): The master node takes the input data, partitions it into smaller subproblems, and distributes individual pieces of the data to worker nodes. A worker node may do this again in turn, leading to a multilevel tree structure. The worker node processes the smaller problem, computes a result, and passes that result back to its master node.

"Reduce" (implemented here in a function `reducefn`): The master node collects the partial results from all the subproblems and combines them in some way to compute the answer to the problem it needs.

Map-Reduce allows for distributed processing of the map and reduction operations. Provided each mapping operation is independent of the others, all maps can be performed in parallel—though in practice, it is limited by the number of independent data sources and/or the number of CPUs near each source. Similarly, a set of "reducers" can perform the reduction phase, provided all outputs of the map operation that share the same key are presented to the same reducer at the same time.

While this process can often appear inefficient compared to algorithms that are more sequential, Map-Reduce can be applied to significantly larger data sets than "commodity" servers can handle—a large server farm can use Map-Reduce to sort a petabyte of data in only a few hours, which would require much longer in a monolithic or single process system.

Parallelism also offers some possibility of recovering from partial failure of servers or storage during the operation: if one mapper or reducer fails, the work can be rescheduled—assuming the input data are still available.

Map-Reduce comprises of two main functions: `mapfn` and `reducefn`. `mapfn` takes a (key,value) pair of data with a type in one data domain and returns a list of (key,value) pairs in a different domain:

$$\text{mapfn}(k1, v1) \rightarrow (k2, v2) \tag{8.14}$$

The `mapfn` function is applied in parallel to every item in the input data set. This produces a list of (k2,v2) pairs for each call. After that, the Map-Reduce framework collects all pairs with the same key from all lists

and groups them together, thus creating one group for each one of the different generated keys. The `reducefn` function is then applied in parallel to each group, which in turn produces a collection of values in the same domain:

$$\text{reducefn}(k2, [\text{list of } v2]) \rightarrow (k2, v3) \tag{8.15}$$

The values returned by `reducefn` are then collected into a single list. Each call to `reducefn` can produce one, none, or multiple partial results. Thus the Map-Reduce framework transforms a list of (key, value) pairs into a list of values. It is necessary but not sufficient to have implementations of the map and reduce abstractions to implement Map-Reduce. Distributed implementations of Map-Reduce require a means of connecting the processes performing the `mapfn` and `reducefn` phases.

Here is a nonparallel implementation that explains the data workflow better:

```
def mapreduce(mapper,reducer,data):
    """
    >>> def mapfn(x): return x%2, 1
    >>> def reducefn(key,values): return len(values)
    >>> data = xrange(100)
    >>> print mapreduce(mapfn,reducefn,data)
    {0: 50, 1: 50}
    """
    partials = {}
    results = {}
    for item in data:
        key,value = mapper(item)
        if not key in partials:
            partials[key]=[value]
        else:
            partials[key].append(value)
    for key,values in partials.items():
        results[key] = reducer(key,values)
    return results
```

And here is an example we can use to find how many random DNA strings contain the subsequence "ACTA":

```
>>> from random import choice
>>> strings = [''.join(choice('ATGC') for i in xrange(10))
...            for j in xrange(100)]
>>> def mapfn(string): return ('ACCA' in string, 1)
>>> def reducefn(check, values): return len(values)
```

```
>>> print mapreduce(mapfn,reducefn,strings)
{False: ..., True: ...}
```

The important thing about the preceding code is that there are two loops in Map-Reduce. Each loop consists of executing tasks (map tasks and reduce tasks) which are independent from each other (all the maps are independent, all the reduce are independent, but the reduce depend on the maps). Because they are independent, they can be executed in parallel and by different processes.

A simple and small library that implements the map-reduce algorithm in Python is `mincemeat` [68]. The workers connect and authenticate to the server using a password and request tasks to executed. The server accepts connections and assigns the map and reduce tasks to the workers.

The communication is performed using asynchronous sockets, which means neither workers nor the master is ever in a wait state. The code is event based, and communication only happens when a socket connecting the master to a worker is ready for a write (task assignment) or a read (task completed).

The code is also failsafe because if a worker closes the connection prematurely, the task is reassigned to another worker.

Function `mincemeat` uses the python libraries `asyncore` and `asynchat` to implement the communication patterns, for which we refer to the Python documentation.

Here is an example of a `mincemeat` program:

```
import mincemeat
from random import choice

strings = [''.join(choice('ATGC') for i in xrange(10)) for j in xrange(100)]
def mapfn(k1, string): yield ('ACCA' in string, 1)
def reducefn(k2, values): return len(values)

s = mincemeat.Server()
s.mapfn = mapfn
s.reducefn = reducefn
s.datasource = dict(enumerate(strings))
results = s.run_server(password='changeme')
print results
```

Notice that in `mincemeat`, the data source is a list of key value dictionaries where the values are the ones to be processed. The key is also passed to the `mapfn` function as first argument. Moreover, the `mapfn` function can return more than one value using `yield`. This syntactical notation makes `minemeat` more flexible.

Execute this script on the server:

```
> python mincemeat_example.py
```

Run mincemeat.py as a worker on a client:

```
> python mincemeat.py -p changeme [server address]
```

You can run more than one worker, although for this example the server will terminate almost immediately.

Function `mincemeat` works fine for many applications, but sometimes one wishes for a more powerful tool that provides faster communications, support for arbitrary languages, and better scalability tools and monitoring tools. An example in Python is `disco`. A standard tool, written in Java but supporting Python, is **Hadoop**.

8.6 pyOpenCL

Nvidia should be credited for bringing GPU computing to the mass market. They have developed the CUDA [69] framework for GPU programming. CUDA programs consist of two parts: a host and a kernel. The host deploys the kernel on the available GPU core, and multiple copies of the kernel run in parallel.

Nvidia, AMD, Intel, and ARM have created the Kronos Group, and together they have developed the Open Common Language framework (OpenCL [70]), which borrows many ideas from CUDA and promises more portability. OpenCL supports Intel/AMD CPUs, Nvidia/ATI GPU, ARM chips, and the LLVM virtual machine.

OpenCL is a C99 dialect. In OpenCL, like in CUDA, there is a host program and a kernel. Multiple copies of the kernel are queued and run in parallel on available devices. Kernels running on the same device have

access to a shared memory area as well as local memory areas.

A typical OpenCL program has the following structure:

```
find available devices (GPUs, CPUs)
copy data from host to device
run N copies of this kernel code on the device
copy data from device to host
```

Usually the kernel is written in C99, while the host is written in C++. It is also possible to write the host code in other languages, including Python. Here we will use the `pyOpenCL` [4] module for programming the host using Python. This produces no significative loss of performance compared to C++ because the actual computation is performed by kernel, not by the host. It is also possible to write the kernels using Python. This can be done using a library called Clyther [71] or one called `ocl` [5]. Here we will use the latter; `ocl` performs a one-time conversion of Python code for the kernel to C99 code. This conversion is done line by line and therefore also introduces no performance loss compared to writing native OpenCL kernels. It also provides an additional abstraction layer on top of `pyOpenCL`, which will make our examples more compact.

8.6.1 A first example with PyOpenCL

`pyOpenCL` uses `numpy` multidimensional arrays to store data. For example, here is a `numpy` example that performs the scalar product between two vectors, u and v:

```
import numpy as npy

size = 10000
u = npy.random.rand(size).astype(npy.float32)
v = npy.random.rand(size).astype(npy.float32)
w = npy.zeros(n,dtype=numpy.float32)

for i in xrange(0, n):
    w[i] = u[i] + v[i];

assert npy.linalg.norm(w - (u + v)) == 0
```

The program works as follows:

- It creates a two `numpy` arrays u and v of given size and filled with ran-

dom numbers.

- It creates another `numpy` array w of the same size filled with zeros.
- It loops over all indices of w and adds, term by term, u and v storing the result into w.
- It checks the result using the `numpy linalg` submodule.

Our goal is to parallelize the part of the computation performed in the loop. Notice that our parallelization will not make the code faster because this is a linear algorithm, and algorithms linear in the input are never faster when parallelized because the communication has the same order of growth as the algorithm itself:

```
from ocl import Device
import numpy as npy

n = 100000
u = npy.random.rand(n).astype(npy.float32)
v = npy.random.rand(n).astype(npy.float32)

device = Device()
u_buffer = device.buffer(source=a)
v_buffer = device.buffer(source=b)
w_buffer = device.buffer(size=b.nbytes)

kernels = device.compile("""
    __kernel void sum(__global const float *u, /* u_buffer  */
                      __global const float *v, /* v_buffer  */
                      __global float *w) {     /* w_buffer  */
      int i = get_global_id(0);                /* thread id */
      w[i] = u[i] + v[i];
    }
    """)

kernels.sum(device.queue,[n],None,u_buffer,v_buffer,w_buffer)
w = device.retrieve(w_buffer)

assert npy.linalg.norm(w - (u+v)) == 0
```

This program performs the following steps in addition to the original non-OpenCL code: it declares a `device` object; it declares a buffer for each of the vectors u, v, and w; it declares and compiles the kernel; it runs the kernel; it retrieves the result.

The `device` object encapsulate the kernel(s) and a queue for kernel submission.

The line:

```
kernels.sum(...,[n],...)
```

submits to the queue `n` instances of the `sum` kernel. Each kernel instance can retrieve its own ID using the function `get_global_id(0)`. Notice that a kernel must be declared with the `__kernel` prefix. Arguments that are to be shared by all kernels must be `__global`.

The `Device` class is defined in the "ocl.py" file in terms of `pyOpenCL` API:

```
import numpy
import pyopencl as pcl

class Device(object):
    flags = pcl.mem_flags
    def __init__(self):
        self.ctx = pcl.create_some_context()
        self.queue = pcl.CommandQueue(self.ctx)
    def buffer(self,source=None,size=0,mode=pcl.mem_flags.READ_WRITE):
        if source is not None: mode = mode|pcl.mem_flags.COPY_HOST_PTR
        buffer = pcl.Buffer(self.ctx,mode, size=size, hostbuf=source)
        return buffer
    def retrieve(self,buffer,shape=None,dtype=numpy.float32):
        output = numpy.zeros(shape or buffer.size/4,dtype=dtype)
        pcl.enqueue_copy(self.queue, output, buffer)
        return output
    def compile(self,kernel):
        return pcl.Program(self.ctx,kernel).build()
```

Here `self.ctx` is the device context, `self.queue` is the device queue. The functions `buffer`, `retrieve`, and `compile` map onto the corresponding pyOpenCL functions `Buffer`, `enqueue_copy`, and `Program` but use a simpler syntax. For more details, we refer to the official `pyOpenCL` documentation.

8.6.2 Laplace solver

In this section we implement a two-dimensional Laplace solver. A three-dimensional generalization is straightforward. In particular, we want to

solve the following differential equation known as a Laplace equation:

$$(\partial_x^2 + \partial_y^2)u(x,y) = q(x,y) \tag{8.16}$$

Here q is the input and u is the output.

This equation originates, for example, in electrodynamics. In this case, q is the distribution of electric charge in space and u is the electrostatic potential.

As we did in chapter 3, we proceed by discretizing the derivatives:

$$\partial_x^2 u(x,y) = (u(x-h,y) - 2u(x,y) + u(x+h,y))/h^2 \tag{8.17}$$
$$\partial_y^2 u(x,y) = (u(x,y-h) - 2u(x,y) + u(x,y+h))/h^2 \tag{8.18}$$

Substitute them into eq. 8.16 and solve the equation in $u(x,y)$. We obtain

$$u(x,y) = 1/4(u(x-h,y) + u(x+h,y) + u(x,y-h) + u(x,y+h) - h^2 q(x,y)) \tag{8.19}$$

We can therefore solve eq. 8.16 by iterating eq. 8.19 until convergence. The initial value of u will not affect the solution, but the closer we can pick it to the actual solution, the faster the convergence.

The procedure we utilized here for transforming a differential equation into an iterative procedure is a general one and applies to other differential equations as well. The iteration proceeds very much as the fixed point solver also examined in chapter 3.

Here is an implementation using `ocl`:

```
from ocl import Device
from canvas import Canvas
from random import randint, choice
import numpy

n = 300
q = numpy.zeros((n,n), dtype=numpy.float32)
u = numpy.zeros((n,n), dtype=numpy.float32)
w = numpy.zeros((n,n), dtype=numpy.float32)

```

```
for k in xrange(n):
    q[randint(1, n-1),randint(1, n-1)] = choice((-1,+1))

device = Device()
q_buffer = device.buffer(source=q, mode=device.flags.READ_ONLY)
u_buffer = device.buffer(source=u)
w_buffer = device.buffer(source=w)


kernels = device.compile("""
       __kernel void solve(__global float *w,
                           __global const float *u,
                           __global const float *q) {
          int x = get_global_id(0);
          int y = get_global_id(1);
          int xy = y*WIDTH + x, up, down, left, right;
          if(y!=0 && y!=WIDTH-1 && x!=0 && x!=WIDTH-1) {
             up=xy+WIDTH; down=xy-WIDTH; left=xy-1; right=xy+1;
             w[xy] = 1.0/4.0*(u[up]+u[down]+u[left]+u[right] - q[xy]);
          }
       }
    """.replace('WIDTH',str(n)))

for k in xrange(1000):
    kernels.solve(device.queue, [n,n], None, w_buffer, u_buffer, q_buffer)
    (u_buffer, w_buffer) = (w_buffer, u_buffer)

u = device.retrieve(u_buffer,shape=(n,n))

Canvas().imshow(u).save(filename='plot.png')
```

We can now use the Python to C99 converter of `ocl` to write the kernel using Python:

```
from ocl import Device
from canvas import Canvas
from random import randint, choice
import numpy

n = 300
q = numpy.zeros((n,n), dtype=numpy.float32)
u = numpy.zeros((n,n), dtype=numpy.float32)
w = numpy.zeros((n,n), dtype=numpy.float32)

for k in xrange(n):
    q[randint(1, n-1),randint(1, n-1)] = choice((-1,+1))

device = Device()
q_buffer = device.buffer(source=q, mode=device.flags.READ_ONLY)
```

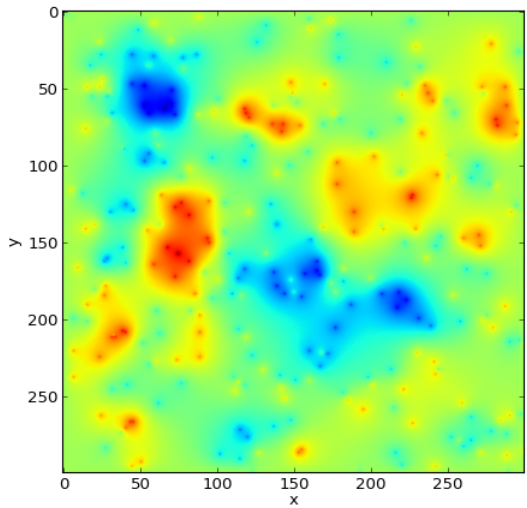

Figure 8.7: The image shows the output of the Laplace program and represents the two-dimensional electrostatic potential for a random charge distribution.

```
u_buffer = device.buffer(source=u)
w_buffer = device.buffer(source=w)

@device.compiler.define_kernel(
    w='global:ptr_float',
    u='global:const:ptr_float',
    q='global:const:ptr_float')
def solve(w,u,q):
    x = new_int(get_global_id(0))
    y = new_int(get_global_id(1))
    xy = new_int(x*n+y)
    if y!=0 and y!=n-1 and x!=0 and x!=n-1:
        up = new_int(xy-n)
        down = new_int(xy+n)
        left = new_int(xy-1)
        right = new_int(xy+1)
        w[xy] = 1.0/4*(u[up]+u[down]+u[left]+u[right] - q[xy])

kernels = device.compile(constants=dict(n=n))

for k in xrange(1000):
    kernels.solve(device.queue, [n,n], None, w_buffer, u_buffer, q_buffer)
    (u_buffer, w_buffer) = (w_buffer, u_buffer)
```

```
39
40 u = device.retrieve(u_buffer,shape=(n,n))
41
42 Canvas().imshow(u).save(filename='plot.png')
```

The output is shown in fig. 8.6.2.

One can pass constants to the kernel using

```
1 device.compile(..., constants = dict(n = n))
```

One can also pass include statements to the kernel:

```
1 device.compile(..., includes = ['#include <math.h>'])
```

where `includes` is a list of `#include` statements.

Notice how the kernel is line by line the same as the original C code. An important part of the new code is the `define_kernel` decorator. It tells `ocl` that the code must be translated to C99. It also declares the type of each argument, for example,

```
1 ...define_kernel(... u='global:const:ptr_float' ...)
```

It means that:

```
1 global const float* u
```

Because in C, one must declare the type of each new variable, we must do the same in `ocl`. This is done using the pseudo-casting operators `new_int`, `new_float`, and so on. For example,

```
1 a = new_int(b+c)
```

is converted into

```
1 int a = b+c;
```

The converter also checks the types for consistency. The return type is determined automatically from the type of the object that is returned. Python objects that have no C99 equivalent like lists, tuples, dictionaries, and sets are not supported. Other types are converted based on the following table:

ocl	C99/OpenCL
`a = new_`*type*`(...)`	*type* `a = ...;`
`a = new_prt_`*type*`(...)`	*type* `*a = ...;`
`a = new_prt_prt_`*type*`(...)`	*type* `**a = ...;`
`None`	`null`
`ADDR(x)`	`&x`
`REFD(x)`	`*x`
`CAST(prt_`*type*`,x)`	`(`*type*`*)x`

8.6.3 Portfolio optimization (in parallel)

In a previous chapter, we provided an algorithm for portfolio optimization. One critical step of that algorithm was the knowledge of all-to-all correlations among stocks. This step can efficiently be performed on a GPU.

In the following example, we solve the same problem again. For each time series `k`, we compute the arithmetic daily returns, `r[k,t]`, and the average returns, `mu[k]`. We then compute the covariance matrix, `cov[i,j]`, and the correlation matrix, `cor[i,j]`. We use different kernels for each part of the computation.

Finally, to make the application more practical, we use MPT [34] to compute a tangency portfolio that maximizes the Sharpe ratio under the assumption of Gaussian returns:

$$\max_x \frac{\mu^T x - r_{\text{free}}}{\sqrt{x^T \Sigma x}} \tag{8.20}$$

Here μ is the vector of average returns (`mu`), Σ is the covariance matrix (`cov`), and r_{free} is the input risk-free interest rate. The tangency portfolio is identified by the vector x (array `x` in the code) whose terms indicate the amount to be invested in each stock (must add up to $1). We perform this maximization on the CPU to demonstrate integration with the `numpy` linear algebra package.

We use the symbols `i` and `j` to identify the stock time series and the

symbol `t` for time (for daily data `t` is a day); `n` is the number of stocks, and `m` is the number of trading days.

We use the `canvas` [11] library, based on the Python `matplotlib` library, to display one of the stock price series and the resulting correlation matrix. Following is the complete code. The output from the code can be seen in fig. 8.6.3.

```
from ocl import Device
from canvas import Canvas
import random
import numpy
from math import exp

n = 1000 # number of time series
m = 250  # number of trading days for time series
p = numpy.zeros((n,m), dtype=numpy.float32)
r = numpy.zeros((n,m), dtype=numpy.float32)
mu = numpy.zeros(n, dtype=numpy.float32)
cov = numpy.zeros((n,n), dtype=numpy.float32)
cor = numpy.zeros((n,n), dtype=numpy.float32)

for k in xrange(n):
    p[k,0] = 100.0
    for t in xrange(1,m):
        c = 1.0 if k==0 else (p[k-1,t]/p[k-1,t-1])
        p[k,t] = p[k,t-1]*exp(random.gauss(0.0001,0.10))*c

device = Device()
p_buffer = device.buffer(source=p, mode=device.flags.READ_ONLY)
r_buffer = device.buffer(source=r)
mu_buffer = device.buffer(source=mu)
cov_buffer = device.buffer(source=cov)
cor_buffer = device.buffer(source=cor)

@device.compiler.define_kernel(p='global:const:ptr_float',
                               r='global:ptr_float')
def compute_r(p, r):
    i = new_int(get_global_id(0))
    for t in xrange(0,m-1):
        r[i*m+t] = p[i*m+t+1]/p[i*m+t] - 1.0

@device.compiler.define_kernel(r='global:ptr_float',
                               mu='global:ptr_float')
def compute_mu(r, mu):
    i = new_int(get_global_id(0))
    sum = new_float(0.0)
```

```
    for t in xrange(0,m-1):
        sum = sum + r[i*m+t]
    mu[i] = sum/(m-1)

@device.compiler.define_kernel(r='global:ptr_float',
       mu='global:ptr_float', cov='global:ptr_float')
def compute_cov(r, mu, cov):
    i = new_int(get_global_id(0))
    j = new_int(get_global_id(1))
    sum = new_float(0.0)
    for t in xrange(0,m-1):
        sum = sum + r[i*m+t]*r[j*m+t]
    cov[i*n+j] = sum/(m-1)-mu[i]*mu[j]

@device.compiler.define_kernel(cov='global:ptr_float',
                               cor='global:ptr_float')
def compute_cor(cov, cor):
    i = new_int(get_global_id(0))
    j = new_int(get_global_id(1))
    cor[i*n+j] = cov[i*n+j] / sqrt(cov[i*n+i]*cov[j*n+j])

program = device.compile(constants=dict(n=n,m=m))

q = device.queue
program.compute_r(q, [n], None, p_buffer, r_buffer)
program.compute_mu(q, [n], None, r_buffer, mu_buffer)
program.compute_cov(q, [n,n], None, r_buffer, mu_buffer, cov_buffer)
program.compute_cor(q, [n,n], None, cov_buffer, cor_buffer)

r = device.retrieve(r_buffer,shape=(n,m))
mu = device.retrieve(mu_buffer,shape=(n,))
cov = device.retrieve(cov_buffer,shape=(n,n))
cor = device.retrieve(cor_buffer,shape=(n,n))

points = [(x,y) for (x,y) in enumerate(p[0])]
Canvas(title='Price').plot(points).save(filename='price.png')
Canvas(title='Correlations').imshow(cor).save(filename='cor.png')

rf = 0.05/m # input daily risk free interest rate
x = numpy.linalg.solve(cov,mu-rf) # cov*x = (mu-rf)
x *= 1.00/sum(x) # assumes 1.00 dollars in total investment
open('optimal_portfolio','w').write(repr(x))
```

Notice how the memory buffers are always one-dimensional, therefore the `i,j` indexes have to be mapped into a one-dimensional index `i*n+j`. Also notice that while kernels `compute_r` and `compute_mu` are called `[n]` times (once per stock `k`), kernels `compute_cov` and `compute_cor` are called `[n,n]`

times, once per each couple of stocks `i,j`. The values of `i,j` are retrieved by `get_global_id(0)` and `(1)`, respectively.

In this program, we have defined multiple kernels and complied them at once. We call one kernel at the time to make sure that the call to the previous kernel is completed before running the next one.

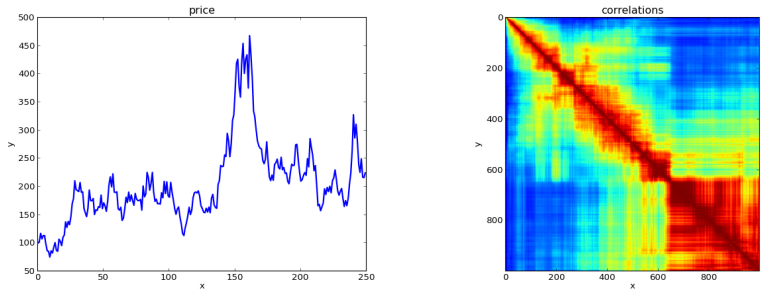

Figure 8.8: The image on the left shows one of the randomly generated stock price histories. The image on the right represents the computed correlation matrix. Rows and columns correspond to stock returns, and the color at the intersection is their correlation (red for high correlation and blue for no correlation). The resulting shape is an artifact of the algorithm used to generate random data.

9
Appendices

9.1 Appendix A: Math Review and Notation

9.1.1 Symbols

∞	infinity
$\wedge$	and
$\vee$	or
$\cap$	intersection
$\cup$	union
$\in$	element or In
$\forall$	for each
$\exists$	exists
$\Rightarrow$	implies
:	such that
iff	if and only if

(9.1)

9.1.2 Set theory

Important sets

$\mathbf{0}$	empty set
$\mathbb{N}$	natural numbers {0,1,2,3,...}
$\mathbb{N}^+$	positive natural numbers {1,2,3,...}
$\mathbb{Z}$	all integers {...,-3,-2,-1,0,1,2,3,...}
$\mathbb{R}$	all real numbers
$\mathbb{R}^+$	positive real numbers (not including 0)
$\mathbb{R}^0$	positive numbers including 0

(9.2)

Set operations

$\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$ are some generic sets.

- **Intersection**

$$\mathcal{A} \cap \mathcal{B} \equiv \{x : x \in \mathcal{A} \text{ and } x \in \mathcal{B}\} \tag{9.3}$$

- **Union**

$$\mathcal{A} \cup \mathcal{B} \equiv \{x : x \in \mathcal{A} \text{ or } x \in \mathcal{B}\} \tag{9.4}$$

- **Difference**

$$\mathcal{A} - \mathcal{B} \equiv \{x : x \in \mathcal{A} \text{ and } x \notin \mathcal{B}\} \tag{9.5}$$

Set laws

- Empty set laws

$$\mathcal{A} \cup \mathbf{0} = \mathcal{A} \tag{9.6}$$
$$\mathcal{A} \cap \mathbf{0} = \mathbf{0} \tag{9.7}$$

- Idempotency laws

$$\mathcal{A} \cup \mathcal{A} = \mathcal{A} \tag{9.8}$$
$$\mathcal{A} \cap \mathcal{A} = \mathcal{A} \tag{9.9}$$

- Commutative laws

$$\mathcal{A} \cup \mathcal{B} = \mathcal{B} \cup \mathcal{A} \tag{9.10}$$
$$\mathcal{A} \cap \mathcal{B} = \mathcal{B} \cap \mathcal{A} \tag{9.11}$$

- Associative laws

$$\mathcal{A} \cup (\mathcal{B} \cup \mathcal{C}) = (\mathcal{A} \cup \mathcal{B}) \cup \mathcal{C} \tag{9.12}$$
$$\mathcal{A} \cap (\mathcal{B} \cap \mathcal{C}) = (\mathcal{A} \cap \mathcal{B}) \cap \mathcal{C} \tag{9.13}$$

- Distributive laws

$$\mathcal{A} \cap (\mathcal{B} \cup \mathcal{C}) = (\mathcal{A} \cap \mathcal{B}) \cup (\mathcal{A} \cap \mathcal{C}) \tag{9.14}$$
$$\mathcal{A} \cup (\mathcal{B} \cap \mathcal{C}) = (\mathcal{A} \cup \mathcal{B}) \cap (\mathcal{A} \cup \mathcal{C}) \tag{9.15}$$

- Absorption laws

$$\mathcal{A} \cap (\mathcal{A} \cup \mathcal{B}) = \mathcal{A} \tag{9.16}$$
$$\mathcal{A} \cup (\mathcal{A} \cap \mathcal{B}) = \mathcal{A} \tag{9.17}$$

- De Morgan's laws

$$\mathcal{A} - (\mathcal{B} \cup \mathcal{C}) = (\mathcal{A} - \mathcal{B}) \cap (\mathcal{A} - \mathcal{C}) \tag{9.18}$$
$$\mathcal{A} - (\mathcal{B} \cap \mathcal{C}) = (\mathcal{A} - \mathcal{B}) \cup (\mathcal{A} - \mathcal{C}) \tag{9.19}$$

More set definitions

- $\mathcal{A}$ is a **subset** of $\mathcal{B}$ iff $\forall x \in \mathcal{A}, x \in \mathcal{B}$
- $\mathcal{A}$ is a **proper subset** of $\mathcal{B}$ iff $\forall x \in \mathcal{A}, x \in \mathcal{B}$ and $\exists x \in \mathcal{B}, x \notin \mathcal{A}$
- $P = \{S_i, i = 1, \ldots, N\}$ (a set of sets S_i) is a **partition** of $\mathcal{A}$ iff $S_1 \cup S_2 \cup \cdots \cup S_N = \mathcal{A}$ and $\forall i, j, S_i \cap S_j = \mathbf{0}$
- The number of elements in a set $\mathcal{A}$ is called the **cardinality** of set $\mathcal{A}$.
- cardinality($\mathbb{N}$)=countable infinite (∞)
- cardinality($\mathbb{R}$)=uncountable infinite (∞) !!!

Relations

- A **Cartesian Product** is defined as

$$\mathcal{A} \times \mathcal{B} = \{(a, b) : a \in \mathcal{A} \text{ and } b \in \mathcal{B}\} \tag{9.20}$$

- A **binary relation** R between two sets $\mathcal{A}$ and $\mathcal{B}$ if a subset of their Cartesian product.
- A binary relation is **transitive** is aRb and bRc implies aRc
- A binary relation is **symmetric** if aRb implies bRa
- A binary relation is **reflexive** if aRa if always true for each a.

Examples:

- $a < b$ for $a \in \mathcal{A}$ and $b \in \mathcal{B}$ is a relation (transitive)
- $a > b$ for $a \in \mathcal{A}$ and $b \in \mathcal{B}$ is a relation (transitive)
- $a = b$ for $a \in \mathcal{A}$ and $b \in \mathcal{B}$ is a relation (transitive, symmetric and reflexive)
- $a \leq b$ for $a \in \mathcal{A}$ and $b \in \mathcal{B}$ is a relation (transitive, and reflexive)
- $a \geq b$ for $a \in \mathcal{A}$ and $b \in \mathcal{B}$ is a relation (transitive, and reflexive)
- A relation R that is transitive, symmetric and reflexive is called an **equivalence relation** and is often indicated with the notation $a \sim b$.

An equivalence relation is the same as a partition.

Functions

- A **function** between two sets $\mathcal{A}$ and $\mathcal{B}$ is a binary relation on $\mathcal{A} \times \mathcal{B}$ and is usually indicated with the notation $f : \mathcal{A} \longmapsto \mathcal{B}$
- The set $\mathcal{A}$ is called **domain** of the function.
- The set $\mathcal{B}$ is called **codomain** of the function.
- A function **maps** each element $x \in \mathcal{A}$ into an element $f(x) = y \in \mathcal{B}$
- The **image** of a function $f : \mathcal{A} \longmapsto \mathcal{B}$ is the set $\mathcal{B}' = \{y \in \mathcal{B} : \exists x \in \mathcal{A}, f(x) = y\} \subseteq \mathcal{B}$

- If $\mathcal{B}'$ is $\mathcal{B}$ then a function is said to be **surjective**.
- If for each x and x' in $\mathcal{A}$ where $x \neq x'$ implies that $f(x) \neq f(x')$ (e.g., if not two different elements of $\mathcal{A}$ are mapped into different element in $\mathcal{B}$) the function is said to be a **bijection**.
- A function $f : \mathcal{A} \longmapsto \mathcal{B}$ is **invertible** if it exists a function $g : \mathcal{B} \longmapsto \mathcal{A}$ such that for each $x \in \mathcal{A}, g(f(x)) = x$ and $y \in \mathcal{B}, f(g(y)) = y$. The function g is indicated with f^{-1}.
- A function $f : \mathcal{A} \longmapsto \mathcal{B}$ is a surjection and a bijection iff f is an invertible function.

Examples:

- $f(n) \equiv n \text{mod} 2$ with domain $\mathbb{N}$ and codomain $\mathbb{N}$ is not a surjection nor a bijection.
- $f(n) \equiv n \text{mod} 2$ with domain $\mathbb{N}$ and codomain $\{0, 1\}$ is a surjection but not a bijection
- $f(x) \equiv 2x$ with domain $\mathbb{N}$ and codomain $\mathbb{N}$ is not a surjection but is a bijection (in fact it is not invertible on odd numbers)
- $f(x) \equiv 2x$ with domain $\mathbb{R}$ and codomain $\mathbb{R}$ is not a surjection and is a bijection (in fact it is invertible)

9.1.3 Logarithms

If $x = a^y$ with $a > 0$, then $y = \log_a x$ with domain $x \in (0, \infty)$ and codomain $y = (-\infty, \infty)$. If the base a is not indicated, the natural log $a = e = 2.7183\ldots$ is assumed.

Properties of logarithms:

$$\log_a x = \frac{\log x}{\log a} \tag{9.21}$$

$$\log xy = (\log x) + (\log y) \tag{9.22}$$

$$\log \frac{x}{y} = (\log x) - (\log y) \tag{9.23}$$

$$\log x^n = n \log x \tag{9.24}$$

9.1.4 Finite sums

Definition

$$\sum_{i=0}^{i<n} f(i) \equiv f(0) + f(1) + \cdots + f(n-1) \tag{9.25}$$

Properties

- **Linearity I**

$$\sum_{i=0}^{i\leq n} f(i) = \sum_{i=0}^{i<n} f(i) + f(n) \tag{9.26}$$

$$\sum_{i=a}^{i\leq b} f(i) = \sum_{i=0}^{i\leq b} f(i) - \sum_{i=0}^{i<a} f(i) \tag{9.27}$$

- **Linearity II**

$$\sum_{i=0}^{i<n} af(i) + bg(i) = a\left(\sum_{i=0}^{i<n} f(i)\right) + b\left(\sum_{i=0}^{i<n} g(i)\right) \tag{9.28}$$

Proof:

$$\begin{aligned}\sum_{i=0}^{i<n} af(i)+bg(i) &= (af(0)+bg(0))+\cdots+(af(n-1)+bg(n-1)) \\ &= af(0)+\cdots+af(n-1)+bg(0)+\cdots+bg(n-1) \\ &= a\left(f(0)+\cdots+f(n-1)\right)+b\left(g(0)+\cdots+g(n-1)\right) \\ &= a\left(\sum_{i=0}^{i<n} f(i)\right)+b\left(\sum_{i=0}^{i<n} g(i)\right) \end{aligned} \tag{9.29}$$

Examples:

$$\sum_{i=0}^{i<n} c = cn \text{ for any constant } c \tag{9.30}$$

$$\sum_{i=0}^{i<n} i = \frac{1}{2}n(n-1) \tag{9.31}$$

$$\sum_{i=0}^{i<n} i^2 = \frac{1}{6}n(n-1)(2n-1) \tag{9.32}$$

$$\sum_{i=0}^{i<n} i^3 = \frac{1}{4}n^2(n-1)^2 \tag{9.33}$$

$$\sum_{i=0}^{i<n} x^i = \frac{x^n-1}{x-1} \text{ (geometric sum)} \tag{9.34}$$

$$\sum_{i=0}^{i<n} \frac{1}{i(i+1)} = 1-\frac{1}{n} \text{ (telescopic sum)} \tag{9.35}$$

9.1.5 Limits ($n \to \infty$)

In these section we will only deal with limits ($n \to \infty$) of positive functions.

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = ? \tag{9.36}$$

First compute limits of the numerator and denominator separately:

$$\lim_{n\to\infty} f(n) = a \tag{9.37}$$

$$\lim_{n\to\infty} g(n) = b \tag{9.38}$$

- If $a \in \mathbb{R}$ and $b \in \mathbb{R}^+$ then

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \frac{a}{b} \tag{9.39}$$

- If $a \in \mathbb{R}$ and $b = \infty$ then

$$\lim_{x\to\infty} \frac{f(x)}{g(x)} = 0 \tag{9.40}$$

- If ($a \in \mathbb{R}^+$ and $b = 0$) or ($a = \infty$ and $b \in \mathbb{R}$)

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty \tag{9.41}$$

- If ($a = 0$ and $b = 0$) or ($a = \infty$ and $b = \infty$) use L'Hôpital's rule

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{f'(n)}{g'(n)} \tag{9.42}$$

  and start again!

- Else ... the limit does not exist (typically oscillating functions or non-analytic functions).

For any $a \in \mathbb{R}$ or $a = \infty$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = a \Rightarrow \lim_{n\to\infty} \frac{g(n)}{f(n)} = 1/a \tag{9.43}$$

Table of derivatives

$f(x)$	$f'(x)$
c	0
ax^n	anx^{n-1}
$\log x$	$\frac{1}{x}$
e^x	e^x
a^x	$a^x \log a$
$x^n \log x, n > 0$	$x^{n-1}(n \log x + 1)$

(9.44)

Practical rules to compute derivatives

$$\frac{d}{dx}\left(f(x) + g(x)\right) = f'(x) + g'(x) \tag{9.45}$$

$$\frac{d}{dx}\left(f(x) - g(x)\right) = f'(x) - g'(x) \tag{9.46}$$

$$\frac{d}{dx}\left(f(x)g(x)\right) = f'(x)g(x) + f(x)g'(x) \tag{9.47}$$

$$\frac{d}{dx}\left(\frac{1}{f(x)}\right) = -\frac{f'(x)}{f(x)^2} \tag{9.48}$$

$$\frac{d}{dx}\left(\frac{f(x)}{g(x)}\right) = \frac{f'(x)}{g(x)} - \frac{f(x)g'(x)}{g(x)^2} \tag{9.49}$$

$$\frac{d}{dx}f(g(x)) = f'(g(x))g'(x) \tag{9.50}$$

Index

Bibliography

[1] http://www.python.org

[2] Travis Oliphant, "A Guide to NumPy". Vol.1. USA: Trelgol Publishing (2006)

[3] http://www.scipy.org/

[4] Andreas Klöckner *et al* "Pycuda and pyopencl: A scripting-based approach to gpu run-time code generation." Parallel Computing 38.3 (2012) pp 157-174

[5] https://github.com/mdipierro/ocl

[6] http://www.network-theory.co.uk/docs/pytut/

[7] http://oreilly.com/catalog/9780596158071

[8] http://www.python.org/doc/

[9] http://www.sqlite.org/

[10] http://matplotlib.sourceforge.net/

[11] https://github.com/mdipierro/canvas

[12] Stefan Behnel *et al.*, "Cython: The best of both worlds." Computing in Science & Engineering 13.2 (2011) pp 31-39

[13] Donald Knuth, "The Art of Computer Programming, Volume 3", Addison-Wesley, (1997). ISBN 0-201-89685-0

[14] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and

Clifford Stein, "Introduction to Algorithms", Second Edition. MIT Press and McGraw-Hill (2001). ISBN 0-262-03293-7

[15] J.W.J. Williams, J. W. J. "Algorithm 232 - Heapsort", Communications of the ACM 7 (6) (1964) pp 347–348

[16] E. F. Moore, "The shortest path through a maze", in Proceedings of the International Symposium on the Theory of Switching, Harvard University Press (1959) pp 285–292

[17] Charles Pierre Trémaux (1859–1882) École Polytechnique of Paris (1876). re-published in the Annals academic, March 2011 – ISSN: 0980-6032

[18] Joseph Kruskal, "On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem", in Proceedings of the American Mathematical Society, Vol.7, N.1 (1956) pp 48–50

[19] R. C. Prim, "Shortest connection networks and some generalizations" in Bell System Technical Journal, 36 (1957) pp 1389–1401

[20] M. Farach-Colton *et al.*, "Mathematical Support for Molecular Biology", DIMACS: Series in Discrete Mathematics and Theoretical Computer Science (1999) Volume 47. ISBN:0-8218-0826-5

[21] B. Korber *et al.*, "Timing the Ancestor of the HIV-1 Pandemic Strains", Science (9 Jun 2000) Vol.288 no.5472.

[22] E. W. Dijkstra, "A note on two problems in connexion with graphs". Numerische Mathematik 1, 269–271 (1959). DOI:10.1007/BF01386390

[23] C. E. Shannon, "A Mathematical Theory of Communication". Bell System Technical Journal 27 (1948) pp 379–423

[24] R. M. Fano, "The transmission of information", Technical Report No. 65 MIT (1949)

[25] D. A. Huffman, "A Method for the Construction of Minimum-Redundancy Codes", Proceedings of the I.R.E., (1952) pp 1098–1102

[26] Bergroth and H. Hakonen and T. Raita, "A Survey of Longest Common Subsequence Algorithms". SPIRE (IEEE Computer Society) 39–48 (200). DOI:10.1109/SPIRE.2000.878178. ISBN:0-7695-0746-8

[27] Saul Needleman and Christian Wunsch, "A general method applicable to the search for similarities in the amino acid sequence of two proteins". Journal of Molecular Biology 48 (3) (1970) pp 443–53. DOI:10.1016/0022-2836(70)90057-4

[28] Tobias Dantzig, "Numbers: The Language of Science", 1930.

[29] V. Estivill-Castro, "Why so many clustering algorithms", ACM SIGKDD Explorations Newsletter 4 (2002) pp 65. DOI:10.1145/568574.568575

[30] J. J. Hopfield, "Neural networks and physical systems with emergent collective computational abilities", Proc. Natl. Acad. Sci. USA Vol. 79 (1982) pp 2554-2558

[31] Nils Aall Barricelli, Nils Aall, "Symbiogenetic evolution processes realized by artificial methods", Methodos (1957) pp 143–182

[32] Michael Garey and David Johnson, "Computers and Intractability: A Guide to the Theory of NP-Completeness", San Francisco: W. H. Freeman and Company (1979). ISBN:0-7167-1045-5

[33] Douglas R. Hofstadter, "Gödel, Escher, Bach: An Eternal Golden Braid", Basic Books 91979). ISBN:0-465-02656-7

[34] Harry Markowitz, "Foundations of portfolio theory", The Journal of Finance 46.2 (2012) pp 469-477

[35] P. E. Greenwood and M. S. Nikulin, "A guide to chi-squared testing". Wiley, New York (1996). ISBN:0-471-55779-X

[36] Andrew Lo and Jasmina Hasanhodzic, "The Evolution of Technical Analysis: Financial Prediction from Babylonian Tablets to Bloomberg Terminals", Bloomberg Press (2010). ISBN:1576603490

[37] Y. Saad and M.H. Schultz, "GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems", SIAM J. Sci. Stat. Comput. 7 (1986). DOI:10.1137/0907058

[38] H. A. Van der Vorst, "Bi-CGSTAB: A Fast and Smoothly Converging Variant of Bi-CG for the Solution of Nonsymmetric Linear Systems". SIAM J. Sci. and Stat. Comput. 13 (2) (1992) pp 631–644. DOI:10.1137/0913035

[39] Richard Burden and Douglas Faires, "2.1 The Bisection Algorithm", Numerical Analysis (3rd ed.), PWS Publishers (1985). ISBN:0-87150-857-5

[40] Michiel Hazewinkel, "Newton method", Encyclopedia of Mathematics, Springer (2001). ISBN:978-1-55608-010-4

[41] Mordecai Avriel and Douglas Wilde, "Optimality proof for the symmetric Fibonacci search technique", Fibonacci Quarterly 4 (1966) pp 265–269 MR:0208812

[42] Loukas Grafakos, "Classical and Modern Fourier Analysis", Prentice-Hall (2004). ISBN:0-13-035399-X

[43] S.D. Poisson, "Probabilité des jugements en matière criminelle et en matière civile, précédées des règles générales du calcul des probabilitiés", Bachelier (1837)

[44] A. W. Van der Vaart, "Asymptotic statistics", Cambridge University Press (1998). ISBN:978-0-521-49603-2

[45] Jonah Lehrer, "How We Decide", Houghton Mifflin Harcourt (2009). ISBN:978-0-618-62011-1

[46] Edward N. Lorenz, "Deterministic non-periodic flow". Journal of the Atmospheric Sciences 20 (2) (1963) pp 130–141. DOI:10.1175/1520-0469

[47] Ian Hacking, "19th-century Cracks in the Concept of Determinism", Journal of the History of Ideas, 44 (3) (1983) pp 455-475 JSTOR:2709176

[48] F. Cannizzaro, G. Greco, S. Rizzo, E. Sinagra, "Results of the measurements carried out in order to verify the validity of the poisson-exponential distribution in radioactive decay events". The International Journal of Applied Radiation and Isotopes 29 (11) (1978) pp 649. DOI:10.1016/0020-708X(78)90101-1

[49] Yuval Perez, "Iterating Von Neumann's Procedure for Extracting Random Bits". The Annals of Statistics 20 (1) (1992) pp 590–597

[50] Martin Luescher, "A Portable High-Quality Random Number Generator for Lattice Field Theory Simulations", Comput. Phys. Commun. 79 (1994) pp 100-110

[51] `http://demonstrations.wolfram.com/PoorStatisticalQualitiesForTheRANDURandomNumberGenerator`

[52] G. S. Fishman, "Grouping observations in digital simulation", Management Science 24 (1978) pp 510-521

[53] P. Good, "Introduction to Statistics Through Resampling Methods and R/S-PLUS", Wiley (2005). ISBN:0-471-71575-1

[54] J. Shao and D. Tu, "The Jackknife and Bootstrap", Springer-Verlag (1995)

[55] Paul Wilmott, "Paul Wilmott Introduces Quantitative Finance", Wiley 92005). ISBN:978-0-470-31958-1

[56] http://www.fas.org/sgp/othergov/doe/lanl/lib-www/la-pubs/00326407.pdf

[57] S. M. Ross, "Stochastic Processes", Wiley (1995). ISBN:978-0-471-12062-9

[58] Révész Pal, "Random walk in random and non random environments", World Scientific (1990)

[59] A.A. Markov. "Extension of the limit theorems of probability theory to a sum of variables connected in a chain". reprinted in Appendix B of R. Howard. "Dynamic Probabilistic Systems",

Vol.1, John Wiley and Sons (1971)

[60] W. Vervaat, "A relation between Brownian bridge and Brownian excursion". Ann. Prob. 7 (1) (1979) pp 143–149 JSTOR:2242845

[61] Kiyoshi Ito, "On stochastic differential equations", Memoirs, American Mathematical Society 4, 1–51 (1951)

[62] Steven Shreve, "Stochastic Calculus for Finance II: Continuous Time Models", Springer (2008) pp 114. ISBN:978-0-387-40101-0

[63] Sorin Istrail and Fumei Lam, "Combinatorial Algorithms for Protein Folding in Lattice Models: A Survey of Mathematical Results" (2009)

[64] Michael Flynn, "Some Computer Organizations and Their Effectiveness". IEEE Trans. Comput. C–21 (9) (1972) pp 948–960. DOI:10.1109/TC.1972.5009071

[65] Gene Amdahl, "Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities", AFIPS Conference Proceedings (30) (1967) pp 483–485

[66] `http://mpi4py.scipy.org/`

[67] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. Commun. ACM 51, 1 (January 2008), 107-113. DOI=10.1145/1327452.1327492 http://doi.acm.org/10.1145/1327452.1327492

[68] `http://remembersaurus.com/mincemeatpy/`

[69] Erik Lindholm *et al.*, "NVIDIA Tesla: A unified graphics and computing architecture." Micro, IEEE 28.2 (2008) pp 39-55.

[70] Aaftab Munshi, "OpenCL: Parallel Computing on the GPU and CPU." SIGGRAPH, Tutorial (2008).

[71] `http://srossross.github.com/Clyther/`