

Random numbers have many applications in science and computer programming, especially when there are significant uncertainties in a phenomenon of interest. The purpose of this chapter is to look at some practical problems involving random numbers and learn how to program with such numbers. We shall make several games and also look into how random numbers can be used in physics. You need to be familiar with the first four chapters in order to study the present chapter, but a few examples and exercises will require familiarity with the class concept from Chapter 7.

The key idea in computer simulations with random numbers is first to formulate an algorithmic description of the phenomenon we want to study. This description frequently maps directly onto a quite simple and short Python program, where we use random numbers to mimic the uncertain features of the phenomenon. The program needs to perform a large number of repeated calculations, and the final answers are “only” approximate, but the accuracy can usually be made good enough for practical purposes. Most programs related to the present chapter produce their results within a few seconds. In cases where the execution times become large, we can vectorize the code. Vectorized computations with random numbers is definitely the most demanding topic in this chapter, but is not mandatory for seeing the power of mathematical modeling via random numbers.

All files associated with the examples in this chapter are found in the folder `src/random`.

## 8.1 Drawing Random Numbers

Python has a module `random` for generating random numbers. The function call `random.random()` generates a random number in the half open interval<sup>1</sup>  $[0, 1)$ . We can try it out:

```
>>> import random
>>> random.random()
0.81550546885338104
>>> random.random()
0.44913326809029852
>>> random.random()
0.88320653116367454
```

All computations of random numbers are based on deterministic algorithms (see Exercise 8.16 for an example), so the sequence of numbers cannot be truly random. However, the sequence of numbers appears to lack any pattern, and we can therefore view the numbers as random<sup>2</sup>.

### 8.1.1 The Seed

Every time we import `random`, the subsequent sequence of `random.random()` calls will yield different numbers. For debugging purposes it is useful to get the same sequence of random numbers every time we run the program. This functionality is obtained by setting a *seed* before we start generating numbers. With a given value of the seed, one and only one sequence of numbers is generated. The seed is an integer and set by the `random.seed` function:

```
>>> random.seed(121)
```

Let us generate two series of random numbers at once, using a list comprehension and a format with two decimals only:

```
>>> random.seed(2)
>>> ['%.2f' % random.random() for i in range(7)]
['0.96', '0.95', '0.06', '0.08', '0.84', '0.74', '0.67']
>>> ['%.2f' % random.random() for i in range(7)]
['0.31', '0.61', '0.61', '0.58', '0.16', '0.43', '0.39']
```

If we set the seed to 2 again, the sequence of numbers is regenerated:

```
>>> random.seed(2)
>>> ['%.2f' % random.random() for i in range(7)]
['0.96', '0.95', '0.06', '0.08', '0.84', '0.74', '0.67']
```

If we do not give a seed, the `random` module sets a seed based on the current time. That is, the seed will be different each time we run the

<sup>1</sup> In the half open interval  $[0, 1)$  the lower limit is included, but the upper limit is not.

<sup>2</sup> What it means to view the numbers as random has fortunately a firm mathematical foundation, so don't let the fact that random numbers are deterministic stop you from using them.

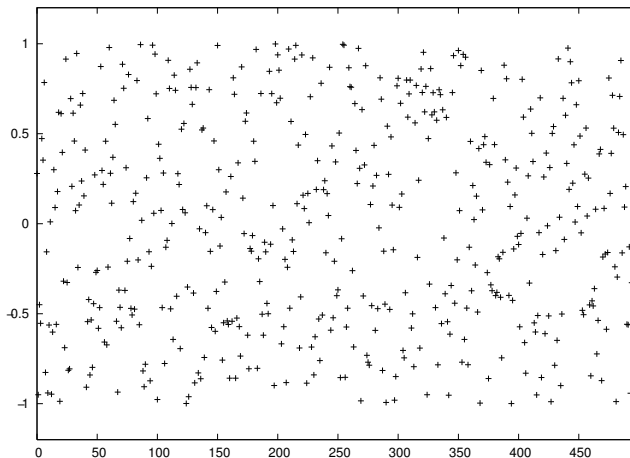
program and consequently the sequence of random numbers will also be different from run to run. This is what we want in most applications. However, we recommend to always set a seed during program development to simplify debugging and verification.

### 8.1.2 Uniformly Distributed Random Numbers

The numbers generated by `random.random()` tend to be equally distributed between 0 and 1, which means that there is no part of the interval  $[0,1)$  with more random numbers than other parts. We say that the distribution of random numbers in this case is *uniform*. The function `random.uniform(a,b)` generates uniform random numbers in the half open interval  $[a,b)$ , where the user can specify  $a$  and  $b$ . With the following program (in file `uniform_numbers0.py`) we may generate lots of random numbers in the interval  $[-1,1)$  and visualize how they are distributed :

```
import random
random.seed(42)
N = 500 # no of samples
x = range(N)
y = [random.uniform(-1,1) for i in x]
from scitools.easyviz import *
plot(x, y, '+', axis=[0,N-1,-1.2,1.2])
```

Figure 8.1 shows the values of these 500 numbers, and as seen, the numbers appear to be random and uniformly distributed between  $-1$  and  $1$ .



**Fig. 8.1** The values of 500 random numbers drawn from the uniform distribution on  $[-1,1)$ .

### 8.1.3 Visualizing the Distribution

It is of interest to see how  $N$  random numbers in an interval  $[a, b]$  are distributed throughout the interval, especially as  $N \rightarrow \infty$ . For example, when drawing numbers from the uniform distribution, we expect that no parts of the interval get more numbers than others. To visualize the distribution, we can divide the interval into subintervals and display how many numbers there are in each subinterval.

Let us formulate this method more precisely. We divide the interval  $[a, b)$  into  $n$  equally sized subintervals, each of length  $h = (b - a)/n$ . These subintervals are called *bins*. We can then draw  $N$  random numbers by calling `random.random()`  $N$  times. Let  $\hat{H}(i)$  be the number of random numbers that fall in bin no.  $i$ ,  $[a + ih, a + (i + 1)h]$ ,  $i = 0, \dots, n - 1$ . If  $N$  is small, the value of  $\hat{H}(i)$  can be quite different for the different bins, but as  $N$  grows, we expect that  $\hat{H}(i)$  varies little with  $i$ .

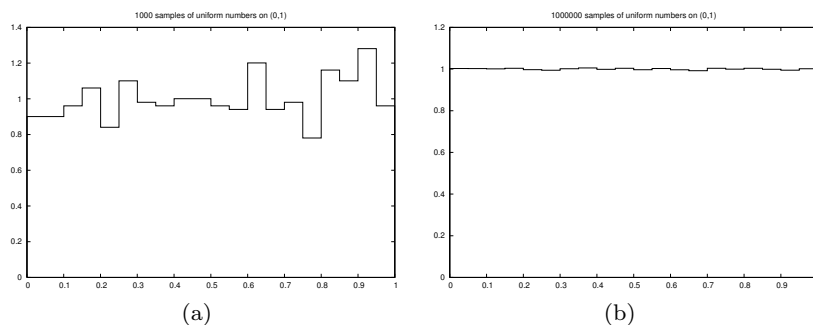
Ideally, we would be interested in how the random numbers are distributed as  $N \rightarrow \infty$  and  $n \rightarrow \infty$ . One major disadvantage is that  $\hat{H}(i)$  increases as  $N$  increases, and it decreases with  $n$ . The quantity  $\hat{H}(i)/N$ , called the frequency count, will reach a finite limit as  $N \rightarrow \infty$ . However,  $\hat{H}(i)/N$  will be smaller and smaller as we increase the number of bins. The quantity  $H(i) = \hat{H}(i)/(Nh)$  reaches a finite limit as  $N, n \rightarrow \infty$ . The probability that a random number lies inside subinterval no.  $i$  is then  $\hat{H}(i)/N = H(i)h$ .

We can visualize  $H(i)$  as a bar diagram (see Figure 8.2), called a *normalized histogram*. We can also define a piecewise constant function  $p(x)$  from  $H(i)$ :  $p(x) = H(i)$  for  $x \in [a + ih, a + (i + 1)h)$ ,  $i = 0, \dots, n - 1$ . As  $n, N \rightarrow \infty$ ,  $p(x)$  approaches the probability density function of the distribution in question. For example, `random.uniform(a,b)` draws numbers from the uniform distribution on  $[a, b)$ , and the probability density function is constant, equal to  $1/(b - a)$ . As we increase  $n$  and  $N$ , we therefore expect  $p(x)$  to approach the constant  $1/(b - a)$ .

The function `compute_histogram` from `scitools.std` returns two arrays `x` and `y` such that `plot(x,y)` plots the piecewise constant function  $p(x)$ . The plot is hence the histogram of the set of random samples. The program below exemplifies the usage:

```
from scitools.std import plot, compute_histogram
import random
samples = [random.random() for i in range(100000)]
x, y = compute_histogram(samples, nbins=20)
plot(x, y)
```

Figure 8.2 shows two plots corresponding to  $N$  taken as  $10^3$  and  $10^6$ . For small  $N$ , we see that some intervals get more random numbers than others, but as  $N$  grows, the distribution of the random numbers becomes more and more equal among the intervals. In the limit  $N \rightarrow \infty$ ,  $p(x) \rightarrow 1$ , which is illustrated by the plot.



**Fig. 8.2** The histogram of uniformly distributed random numbers in 20 bins.

### 8.1.4 Vectorized Drawing of Random Numbers

There is a `random` module in the Numerical Python package which can be used to efficiently draw a possibly large array of random numbers:

```
from numpy import random
r = random.random()           # one number between 0 and 1
r = random.random(size=10000) # array with 10000 numbers
r = random.uniform(-1, 10)    # one number between -1 and 10
r = random.uniform(-1, 10, size=10000) # array
```

There are thus two `random` modules to be aware of: one in the standard Python library and one in `numpy`. For drawing uniformly distributed numbers, the two `random` modules have the same interface, except that the functions from `numpy`'s `random` module has an extra `size` parameter. Both modules also have a `seed` function for fixing the seed.

Vectorized drawing of random numbers using `numpy`'s `random` module is efficient because all the numbers are drawn “at once” in fast C code. You can measure the efficiency gain with the `time.clock()` function as explained on page 447 and in Appendix E.6.1.

*Warning.* It is easy to do an `import random` followed by a `from scitools.std import *` or a `from numpy import *` without realizing that the latter two import statements import a name `random` that overwrites the same name that was imported in `import random`. The result is that the effective `random` module becomes the one from `numpy`. A possible solution to this problem is to introduce a different name for Python's `random` module:

```
import random as random_number
```

We will use this convention in the rest of the book. When you see only the word `random` you then know that this is `numpy.random`.

### 8.1.5 Computing the Mean and Standard Deviation

You probably know the formula for the mean or average of a set of  $n$  numbers  $x_0, x_1, \dots, x_{n-1}$ :

$$x_m = \frac{1}{n} \sum_{j=0}^{n-1} x_j. \quad (8.1)$$

The amount of spreading of the  $x_i$  values around the mean  $x_m$  can be measured by the *variance*<sup>3</sup>,

$$x_v = \frac{1}{n} \sum_{j=0}^{n-1} (x_j - x_m)^2. \quad (8.2)$$

A variant of this formula reads

$$x_v = \frac{1}{n} \left( \sum_{j=0}^{n-1} x_j^2 \right) - x_m^2. \quad (8.3)$$

The good thing with this latter formula is that one can, as a statistical experiment progresses and  $n$  increases, record the sums

$$s_m = \sum_{j=0}^{q-1} x_j, \quad s_v = \sum_{j=0}^{q-1} x_j^2 \quad (8.4)$$

and then, when desired, efficiently compute the most recent estimate on the mean value and the variance after  $q$  samples by

$$x_m = s_m/q, \quad x_v = s_v/q - s_m^2/q^2. \quad (8.5)$$

The *standard deviation*

$$x_s = \sqrt{x_v} \quad (8.6)$$

is often used as an alternative to the variance, because the standard deviation has the same unit as the measurement itself. A common way to express an uncertain quantity  $x$ , based on a data set  $x_0, \dots, x_{n-1}$ , from simulations or physical measurements, is  $x_m \pm x_s$ . This means that  $x$  has an uncertainty of one standard deviation  $x_s$  to either side of the mean value  $x_m$ . With probability theory and statistics one can provide many other, more precise measures of the uncertainty, but that is the topic of a different course.

Below is an example where we draw numbers from the uniform distribution on  $[-1, 1)$  and compute the evolution of the mean and standard

<sup>3</sup> Textbooks in statistics teach you that it is more appropriate to divide by  $n - 1$  instead of  $n$ , but we are not going to worry about that fact in this book.

deviation 10 times during the experiment, using the formulas (8.1) and (8.3)–(8.6):

```
import sys
N = int(sys.argv[1])
import random as random_number
from math import sqrt
sm = 0; sv = 0
for q in range(1, N+1):
    x = random_number.uniform(-1, 1)
    sm += x
    sv += x**2

    # write out mean and st.dev. 10 times in this loop:
    if q % (N/10) == 0:
        xm = sm/q
        xs = sqrt(sv/q - xm**2)
        print '%10d mean: %12.5e  stdev: %12.5e' % (q, xm, xs)
```

The if test applies the *mod* function,  $a \% b$  is 0 if  $b$  times an integer equals  $a$ . The particular if test here is true when  $i$  equals 0,  $N/10$ ,  $2*N/10$ , ...,  $N$ , i.e., 10 times during the execution of the loop. The program is available in the file `mean_stdev_uniform1.py`. A run with  $N = 10^6$  gives the output

```
100000 mean:  1.86276e-03  stdev:  5.77101e-01
200000 mean:  8.60276e-04  stdev:  5.77779e-01
300000 mean:  7.71621e-04  stdev:  5.77753e-01
400000 mean:  6.38626e-04  stdev:  5.77944e-01
500000 mean: -1.19830e-04  stdev:  5.77752e-01
600000 mean:  4.36091e-05  stdev:  5.77809e-01
700000 mean: -1.45486e-04  stdev:  5.77623e-01
800000 mean:  5.18499e-05  stdev:  5.77633e-01
900000 mean:  3.85897e-05  stdev:  5.77574e-01
1000000 mean: -1.44821e-05  stdev:  5.77616e-01
```

We see that the mean is getting smaller and approaching zero as expected since we generate numbers between  $-1$  and  $1$ . The theoretical value of the standard deviation, as  $N \rightarrow \infty$ , equals  $\sqrt{1/3} \approx 0.57735$ .

We have also made a corresponding vectorized version of the code above using `numpy.random` and the ready-made functions `numpy.mean`, `numpy.var`, and `numpy.std` for computing the mean, variance, and standard deviation (respectively) of an array of numbers:

```
import sys
N = int(sys.argv[1])
from numpy import random, mean, var, std, sqrt
x = random.uniform(-1, 1, size=N)
xm = mean(x)
xv = var(x)
xs = std(x)
print '%10d mean: %12.5e  stdev: %12.5e' % (N, xm, xs)
```

This program can be found in the file `mean_stdev_uniform2.py`.

### 8.1.6 The Gaussian or Normal Distribution

In some applications we want random numbers to cluster around a specific value  $m$ . This means that it is more probable to generate a

number close to  $m$  than far away from  $m$ . A widely used distribution with this qualitative property is the Gaussian or normal distribution<sup>4</sup>. The normal distribution has two parameters: the mean value  $m$  and the standard deviation  $s$ . The latter measures the width of the distribution, in the sense that a small  $s$  makes it less likely to draw a number far from the mean value, and a large  $s$  makes more likely to draw a number far from the mean value.

Single random numbers from the normal distribution can be generated by

```
import random as random_number
r = random_number.normalvariate(m, s)
```

while efficient generation of an array of length  $N$  is enabled by

```
from numpy import random
r = random.normal(m, s, size=N)
```

The following program draws  $N$  random numbers from the normal distribution in a loop, computes the mean and standard deviation, and plots the histogram:

```
N = int(sys.argv[1])
m = float(sys.argv[2])
s = float(sys.argv[3])

import random as random_number
random_number.seed(12) # for debugging/testing
from scitools.std import *

samples = [random_number.normalvariate(m, s) for i in range(N)]
x, y = compute_histogram(samples, 20, piecewise_constant=True)

print mean(samples), std(samples)
plot(x, y)
title('%d samples of Gaussian random numbers on (0,1)' % N)
hardcopy('tmp.eps')
```

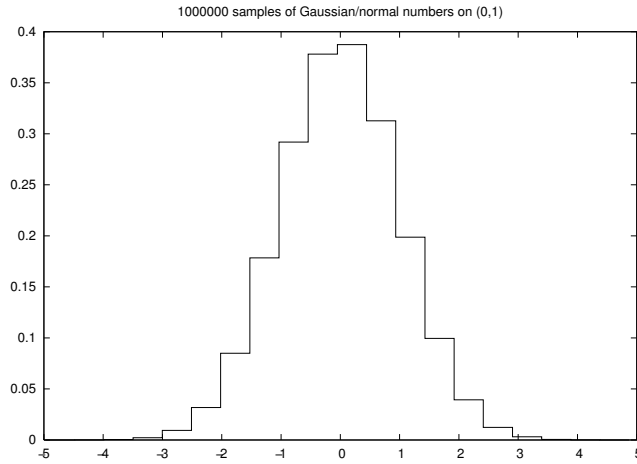
The corresponding program file is `normal_numbers1.py`, which gives a mean of  $-0.00253$  and a standard deviation of  $0.99970$  when run with  $N$  as 1 million,  $m$  as 0, and  $s$  equal to 1. Figure 8.3 shows that the random numbers cluster around the mean  $m = 0$  in a histogram. This normalized histogram will, as  $N$  goes to infinity, approach a bell-shaped function, known as the normal distribution probability density function, given in (1.6) on page 45.

## 8.2 Drawing Integers

Suppose we want to draw a random integer among the values 1, 2, 3, and 4, and that each of the four values is equally probable. One

<sup>4</sup> For example, the blood pressure among adults of one gender has values that follow a normal distribution.





**Fig. 8.3** Normalized histogram of 1 million random numbers drawn from the normal distribution.

possibility is to draw real numbers from the uniform distribution on, e.g.,  $[0, 1)$  and divide this interval into four equal subintervals:

```
import random as random_number
r = random_number.random()
if 0 <= r < 0.25:
    r = 1
elif 0.25 <= r < 0.5:
    r = 2
elif 0.5 <= r < 0.75:
    r = 3
else:
    r = 4
```

Nevertheless, the need for drawing uniformly distributed integers occurs quite frequently, so there are special functions for returning random integers in a specified interval  $[a, b]$ .

### 8.2.1 Random Integer Functions

Python's `random` module has a built-in function `randint(a,b)` for drawing an integer in  $[a, b]$ , i.e., the return value is among the numbers  $a, a+1, \dots, b-1, b$ .

```
import random as random_number
r = random_number.randint(a, b)
```

The `numpy.random.randint(a, b, N)` function has a similar functionality for vectorized drawing of an array of length  $N$  of random integers in  $[a, b)$ . The upper limit  $b$  is not among the drawn numbers, so if we want to draw from  $a, a+1, \dots, b-1, b$ , we must write

```
from numpy import random
r = random.randint(a, b+1, N)
```

Another function, `random_integers(a, b, N)`, also in `numpy.random`, includes the upper limit `b` in the possible set of random integers:

```
from numpy import random
r = random.random_integers(a, b, N)
```

### 8.2.2 Example: Throwing a Die

We can make a program that lets the computer throw a die `N` times and count how many times we get six eyes:

```
import random as random_number
import sys
N = int(sys.argv[1]) # perform N experiments
M = 0               # no of times we get 6 eyes
for i in xrange(N):
    outcome = random_number.randint(1, 6)
    if outcome == 6:
        M += 1
print 'Got six %d times out of %d' % (M, N)
```

We use `xrange` instead of `range` because the former is more efficient when `N` is large (see remark in Exercise 2.46). The vectorized version of this code can be expressed as follows:

```
from numpy import random, sum
import sys
N = int(sys.argv[1])
eyes = random.randint(1, 7, N)
success = eyes == 6 # True/False array
M = sum(success)    # treats True as 1, False as 0
print 'Got six %d times out of %d' % (M, N)
```

The `eyes == 6` construction results in an array with `True` or `False` values, and `sum` applied to this array treats `True` as 1 and `False` as 0 (the integer equivalents to the boolean values), so the sum is the number of elements in `eyes` that equals 6. A very important point here for computational efficiency is to use `sum` from `numpy` and not the standard `sum` function that is available in standard Python. With the former `sum` function, the vectorized version runs about 50 times faster than the scalar version. (With the standard `sum` function in Python, the vectorized versions is in fact slower than the scalar version.)

The two small programs above are found in the files `roll_die.py` and `roll_die_vec.py`, respectively. You can try the programs and see how much faster the vectorized version is (`N` probably needs to be of size at least  $10^6$  to see any noticeable differences for practical purposes).

### 8.2.3 Drawing a Random Element from a List

Given a list `a`, the statement

```
re = random_number.choice(a)
```

picks out an element of `a` at random, and `re` refers to this element. The shown call to `random_number.choice` is the same as

```
re = a[random_number.randint(0, len(a)-1)]
```

There is also a function `shuffle` that permutes the list elements in a random order:

```
random_number.shuffle(a)
```

Picking now `a[0]`, for instance, has the same effect as `random.choice` on the original, unshuffled list. Note that `shuffle` changes the list given as argument.

The `numpy.random` module has also a `shuffle` function with the same functionality.

A small session illustrates the various methods for picking a random element from a list:

```
>>> awards = ['car', 'computer', 'ball', 'pen']
>>> import random as random_number
>>> random_number.choice(awards)
'car'
>>> awards[random_number.randint(0, len(awards)-1)]
'pen'
>>> random_number.shuffle(awards)
>>> awards[0]
'computer'
```

### 8.2.4 Example: Drawing Cards from a Deck

The following function creates a deck of cards, where each card is represented as a string, and the deck is a list of such strings:

```
def make_deck():
    ranks = ['A', '2', '3', '4', '5', '6', '7',
             '8', '9', '10', 'J', 'Q', 'K']
    suits = ['C', 'D', 'H', 'S']
    deck = []
    for s in suits:
        for r in ranks:
            deck.append(s + r)
    random_number.shuffle(deck)
    return deck
```

Here, 'A' means an ace, 'J' represents a jack, 'Q' represents a queen, 'K' represents a king, 'C' stands for clubs, 'D' stands for diamonds, 'H' means hearts, and 'S' means spades. The computation of the list

deck can alternatively (and more compactly) be done by a one-line list comprehension:

```
deck = [s+r for s in suits for r in ranks]
```

We can draw a card at random by

```
deck = make_deck()
card = deck[0]
del deck[0]
# or better:
card = deck.pop(0) # return and remove element with index 0
```

Drawing a hand of *n* cards from a shuffled deck is accomplished by

```
def deal_hand(n, deck):
    hand = [deck[i] for i in range(n)]
    del deck[:n]
    return hand, deck
```

Note that we must return *deck* to the calling code since this list is changed. Also note that the *n* first cards of the deck are random cards if the deck is shuffled (and any deck made by `make_deck` is shuffled).

The following function deals cards to a set of players:

```
def deal(cards_per_hand, no_of_players):
    deck = make_deck()
    hands = []
    for i in range(no_of_players):
        hand, deck = deal_hand(cards_per_hand, deck)
        hands.append(hand)
    return hands
```

```
players = deal(5, 4)
import pprint; pprint.pprint(players)
```

The *players* list may look like

```
[['D4', 'CQ', 'H10', 'DK', 'CK'],
 ['D7', 'D6', 'SJ', 'S4', 'C5'],
 ['C3', 'DQ', 'S3', 'C9', 'DJ'],
 ['H6', 'H9', 'C6', 'D5', 'S6']]
```

The next step is to analyze a hand. Of particular interest is the number of pairs, three of a kind, four of a kind, etc. That is, how many combinations there are of *n\_of\_a\_kind* cards of the same rank (e.g., *n\_of\_a\_kind*=2 finds the number of pairs):

```
def same_rank(hand, n_of_a_kind):
    ranks = [card[1:] for card in hand]
    counter = 0
    already_counted = []
    for rank in ranks:
        if rank not in already_counted and \
            ranks.count(rank) == n_of_a_kind:
            counter += 1
            already_counted.append(rank)
    return counter
```

Note how convenient the `count` method in list objects is for counting how many copies there are of one element in the list.

Another analysis of the hand is to count how many cards there are of each suit. A dictionary with the suit as key and the number of cards with that suit as value, seems appropriate to return. We pay attention only to suits that occur more than once:

```
def same_suit(hand):
    suits = [card[0] for card in hand]
    counter = {} # counter[suit] = how many cards of suit
    for suit in suits:
        count = suits.count(suit)
        if count > 1:
            counter[suit] = count
    return counter
```

For a set of players we can now analyze their hands:

```
for hand in players:
    print """\
The hand %s
has %d pairs, %s 3-of-a-kind and
%s cards of the same suit."" % \
    ('', '.join(hand), same_rank(hand, 2),
     same_rank(hand, 3),
     '+' .join([str(s) for s in same_suit(hand).values()]))
```

The values we feed into the `printf` string undergo some massage: we join the card values with comma and put a plus in between the counts of cards with the same suit. (The `join` function requires a string argument. That is why the integer counters of cards with the same suit, returned from `same_suit`, must be converted to strings.) The output of the `for` loop becomes

```
The hand D4, CQ, H10, DK, CK
has 1 pairs, 0 3-of-a-kind and
2+2 cards of the same suit.
The hand D7, D6, SJ, S4, C5
has 0 pairs, 0 3-of-a-kind and
2+2 cards of the same suit.
The hand C3, DQ, S3, C9, DJ
has 1 pairs, 0 3-of-a-kind and
2+2 cards of the same suit.
The hand H6, H9, C6, D5, S6
has 0 pairs, 1 3-of-a-kind and
2 cards of the same suit.
```

The file `cards.py` contains the functions `make_deck`, `hand`, `hand2`, `same_rank`, `same_suit`, and the test snippets above. With the `cards.py` file one can start to implement real card games.

### 8.2.5 Example: Class Implementation of a Deck

To work with a deck of cards with the code from the previous section one needs to shuffle a global variable `deck` in and out of functions. A set of functions that update global variables (like `deck`) is a primary candidate for a class: The global variables are stored as attributes and

the functions become class methods. This means that the code from the previous section is better implemented as a class. We introduce class `Deck` with a list of cards, `deck`, as attribute, and methods for dealing one or several hands and for putting back a card:

```
class Deck:
    def __init__(self):
        ranks = ['A', '2', '3', '4', '5', '6', '7',
                 '8', '9', '10', 'J', 'Q', 'K']
        suits = ['C', 'D', 'H', 'S']
        self.deck = [s+r for s in suits for r in ranks]
        random_number.shuffle(self.deck)

    def hand(self, n=1):
        """Deal n cards. Return hand as list."""
        hand = [self.deck[i] for i in range(n)] # pick cards
        del self.deck[:n]                       # remove cards
        return hand

    def deal(self, cards_per_hand, no_of_players):
        """Deal no_of_players hands. Return list of lists."""
        return [self.hand(cards_per_hand) \
                for i in range(no_of_players)]

    def putback(self, card):
        """Put back a card under the rest."""
        self.deck.append(card)

    def __str__(self):
        return str(self.deck)
```

This class is found in the module file `Deck.py`. Dealing a hand of five cards to `p` players is coded as

```
from Deck import Deck
deck = Deck()
print deck
players = deck.deal(5, 4)
```

Here, `players` become a nested list as shown in Chapter 8.2.4.

One can go a step further and make more classes for assisting card games. For example, a card has so far been represented by a plain string, but we may well put that string in a class `Card`:

```
class Card:
    """Representation of a card as a string (suit+rank)."""
    def __init__(self, suit, rank):
        self.card = suit + str(rank)

    def __str__(self): return self.card
    def __repr__(self): return str(self)
```

Note that `str(self)` is equivalent to `self.__str__()`.

A `Hand` contains a set of `Card` instances and is another natural abstraction, and hence a candidate for a class:

```
class Hand:
    """Representation of a hand as a list of Card objects."""
    def __init__(self, list_of_cards):
        self.hand = list_of_cards
```

```
def __str__(self):    return str(self.hand)
def __repr__(self):  return str(self)
```

With the aid of classes `Card` and `Hand`, class `Deck` can be reimplemented as

```
class Deck:
    """Representation of a deck as a list of Card objects."""
    def __init__(self):
        ranks = ['A', '2', '3', '4', '5', '6', '7',
                  '8', '9', '10', 'J', 'Q', 'K']
        suits = ['C', 'D', 'H', 'S']
        self.deck = [Card(s,r) for s in suits for r in ranks]
        random_number.shuffle(self.deck)

    def hand(self, n=1):
        """Deal n cards. Return hand as a Hand object."""
        hand = Hand([self.deck[i] for i in range(n)])
        del self.deck[:n]      # remove cards
        return hand

    def deal(self, cards_per_hand, no_of_players):
        """Deal no_of_players hands. Return list of Hand obj."""
        return [self.hand(cards_per_hand) \
                for i in range(no_of_players)]

    def putback(self, card):
        """Put back a card under the rest."""
        self.deck.append(card)

    def __str__(self):
        return str(self.deck)

    def __repr__(self):
        return str(self)

    def __len__(self):
        return len(self.deck)
```

The module file `Deck2.py` contains this implementation. The usage of the two `Deck` classes is the same,

```
from Deck2 import Deck
deck = Deck()
players = deck.deal(5, 4)
```

with the exception that `players` in the last case holds a list of `Hand` instances, and each `Hand` instance holds a list of `Card` instances.

We stated in Chapter 7.3.9 that the `__repr__` method should return a string such that one can recreate the object from this string by the aid of `eval`. However, we did not follow this rule in the implementation of classes `Card`, `Hand`, and `Deck`. Why? The reason is that we want to print a `Deck` instance. Python's `print` or `pprint` on a list applies `repr(e)` to print an element `e` in the list. Therefore, if we had implemented

```
class Card:
    ...
    def __repr__(self):
        return "Card('%s', %s)" % (self.card[0], self.card[1:])
```

```
class Hand:
    ...
    def __repr__(self): return 'Hand(%s)' % repr(self.hand)
```

a plain printing of the deck list of Hand instances would lead to output like

```
[Hand([Card('C', '10'), Card('C', '4'), Card('H', 'K'), ...]),
 Hand([Card('D', '7'), Card('C', '5'), ..., Card('D', '9')])]
```

This output is harder to read than

```
[[C10, C4, HK, DQ, HQ],
 [SA, S8, H3, H10, C2],
 [HJ, C7, S2, CQ, DK],
 [D7, C5, DJ, S3, D9]]
```

That is why we let `__repr__` in classes `Card` and `Hand` return the same “pretty print” string as `__str__`, obtained by returning `str(self)`.

### 8.3 Computing Probabilities

With the mathematical rules from *probability theory* one may compute the probability that a certain event happens, say the probability that you get one black ball when drawing three balls from a hat with four black balls, six white balls, and three green balls. Unfortunately, theoretical calculations of probabilities may soon become hard or impossible if the problem is slightly changed. There is a simple “numerical way” of computing probabilities that is generally applicable to problems with uncertainty. The principal ideas of this approximate technique is explained below, followed by with three examples of increasing complexity.

#### 8.3.1 Principles of Monte Carlo Simulation

Assume that we perform  $N$  experiments where the outcome of each experiment is random. Suppose that some event takes place  $M$  times in these  $N$  experiments. An estimate of the probability of the event is then  $M/N$ . The estimate becomes more accurate as  $N$  is increased, and the exact probability is assumed to be reached in the limit as  $N \rightarrow \infty$ . (Note that in this limit,  $M \rightarrow \infty$  too, so for rare events, where  $M$  may be small in a program, one must increase  $N$  such that  $M$  is sufficiently large for  $M/N$  to become a good approximation to the probability.)

Programs that run a large number of experiments and record the outcome of events are often called *simulation programs*<sup>5</sup>. The mathematical technique of letting the computer perform lots of experiments

<sup>5</sup> This term is also applied for programs that solve equations arising in mathematical models in general, but it is particularly common to use the term when random numbers are used to estimate probabilities.



based on drawing random numbers is commonly called *Monte Carlo simulation*. This technique has proven to be extremely useful throughout science and industry in problems where there is uncertain or random behavior is involved<sup>6</sup>. For example, in finance the stock market has a random variation that must be taken into account when trying to optimize investments. In offshore engineering, environmental loads from wind, currents, and waves show random behavior. In nuclear and particle physics, random behavior is fundamental according to quantum mechanics and statistical physics. Many probabilistic problems can be calculated exactly by mathematics from probability theory, but very often Monte Carlo simulation is the only way to solve statistical problems. Chapters 8.3.2–8.3.4 applies examples to explain the essence of Monte Carlo simulation in problems with inherent uncertainty. However, also deterministic problems, such as integration of functions, can be computed by Monte Carlo simulation (see Chapter 8.5).

### 8.3.2 Example: Throwing Dice

What is the probability of getting at least six eyes twice when rolling four dice? The experiment is to roll four dice, and the event we are looking for appears when we get two or more dice with six eyes. A program `roll_dice1.py` simulating  $N$  such experiments may look like this:

```
import random as random_number
import sys
N = int(sys.argv[1]) # no of experiments
M = 0                # no of successful events
for i in range(N):
    six = 0           # count the no of dice with a six
    r1 = random_number.randint(1, 6)
    if r1 == 6:
        six += 1
    r2 = random_number.randint(1, 6)
    if r2 == 6:
        six += 1
    r3 = random_number.randint(1, 6)
    if r3 == 6:
        six += 1
    r4 = random_number.randint(1, 6)
    if r4 == 6:
        six += 1
    # successful event?
    if six >= 2:
        M += 1
p = float(M)/N
print 'probability:', p
```

*Generalization.* We can easily parameterize how many dice (`ndice`) we roll in each experiment and how many dice with six eyes we want to see

<sup>6</sup> “As far as the laws of mathematics refer to reality, they are not certain, as far as they are certain, they do not refer to reality.” –Albert Einstein, physicist, 1879-1955.

(`nsix`). Thereby, we get a shorter and more general code. The increased generality usually makes it easier to apply or adapt to new problems. The resulting program is found in `roll_dice2.py` and is listed below:

```
import random as random_number
import sys
N = int(sys.argv[1])      # no of experiments
ndice = int(sys.argv[2])  # no of dice
nsix = int(sys.argv[3])   # wanted no of dice with six eyes
M = 0                    # no of successful events
for i in range(N):
    six = 0               # how many dice with six eyes?
    for j in range(ndice):
        # roll die no. j:
        r = random_number.randint(1, 6)
        if r == 6:
            six += 1
    # successful event?
    if six >= nsix:
        M += 1
p = float(M)/N
print 'probability:', p
```

With this program we may easily change the problem setting and ask for the probability that we get six eyes  $q$  times when we roll  $q$  dice. The theoretical probability can be calculated to be  $6^{-4} \approx 0.00077$ , and a program performing  $10^5$  experiments estimates this probability to 0.0008. For such small probabilities the number of successful events  $M$  is small, and  $M/N$  will not be a good approximation to the probability unless  $M$  is reasonably large, which requires a very large  $N$ . The `roll_dice2.py` program runs quite slowly for one million experiments, so it is a good idea to try to vectorize the code to speed up the experiments. Unfortunately, this may constitute a challenge for newcomers to programming, as shown below.

*Vectorization.* In a vectorized version of the `roll_dice2.py` program, we generate a two-dimensional array of random numbers where the first dimension reflects the experiments and the second dimension reflects the trials in each experiment:

```
from numpy import random, sum
eyes = random.randint(1, 7, (N, ndice))
```

The next step is to count the number of successes in each experiment. For this purpose, we must avoid explicit loops if we want the program to run fast. In the present example, we can compare all rolls with 6, resulting in an array `compare` (dimension as `eyes`) with ones for rolls with 6 and 0 otherwise. Summing up the rows in `compare`, we are interested in the rows where the sum is equal to or greater than `nsix`. The number of such rows equals the number of successful events,

which we must divide by the total number of experiments to get the probability<sup>7</sup>:

```
compare = eyes == 6
nthrows_with_6 = sum(compare, axis=1) # sum over columns (index 1)
nsuccesses = nthrows_with_6 >= nsix
M = sum(nsuccesses)
p = float(M)/N
```

The complete program is found in the file `roll_dice2_vec.py`. Getting rid of the two loops, as we obtained in the vectorized version, speeds up the probability estimation with a factor of 40. However, the vectorization is highly non-trivial, and the technique depends on details of how we define success of an event in an experiment.

### 8.3.3 Example: Drawing Balls from a Hat

Suppose there are 12 balls in a hat: four black, four red, and four blue. We want to make a program that draws three balls at random from the hat. It is natural to represent the collection of balls as a list. Each list element can be an integer 1, 2, or 3, since we have three different types of balls, but it would be easier to work with the program if the balls could have a color instead of an integer number. This is easily accomplished by defining color names:

```
colors = 'black', 'red', 'blue' # (tuple of strings)
hat = []
for color in colors:
    for i in range(4):
        hat.append(color)
```

Drawing a ball at random is performed by

```
import random as random_number
color = random_number.choice(hat)
print color
```

Drawing  $n$  balls without replacing the drawn balls requires us to remove an element from the hat when it is drawn. There are three ways to implement the procedure: (i) we perform a `hat.remove(color)`, (ii) we draw a random index with `randint` from the set of legal indices in the `hat` list, and then we do a `del hat[index]` to remove the element, or (iii) we can compress the code in (ii) to `hat.pop(index)`.

```
def draw_ball(hat):
    color = random_number.choice(hat)
    hat.remove(color)
```

<sup>7</sup> This code is considered advanced so don't be surprised if you don't understand much of it. A first step toward understanding is to type in the code and write out the individual arrays for (say)  $N = 2$ . The use of `numpy`'s `sum` function is essential for efficiency.

```

        return color, hat

def draw_ball(hat):
    index = random_number.randint(0, len(hat)-1)
    color = hat[index]
    del hat[index]
    return color, hat

def draw_ball(hat):
    index = random_number.randint(0, len(hat)-1)
    color = hat.pop(index)
    return color, hat

# draw n balls from the hat:
balls = []
for i in range(n):
    color, hat = draw_ball(hat)
    balls.append(color)
print 'Got the balls', balls

```

We can extend the experiment above and ask the question: What is the probability of drawing two or more black balls from a hat with 12 balls, four black, four red, and four blue? To this end, we perform  $N$  experiments, count how many times  $M$  we get two or more black balls, and estimate the probability as  $M/N$ . Each experiment consists of making the `hat` list, drawing a number of balls, and counting how many black balls we got. The latter task is easy with the `count` method in list objects: `hat.count('black')` counts how many elements with value 'black' we have in the list `hat`. A complete program for this task is listed below. The program appears in the file `balls_in_hat.py`.

```

import random as random_number

def draw_ball(hat):
    """Draw a ball using list index."""
    index = random_number.randint(0, len(hat)-1)
    color = hat.pop(index)
    return color, hat

def draw_ball(hat):
    """Draw a ball using list index."""
    index = random_number.randint(0, len(hat)-1)
    color = hat[index]
    del hat[index]
    return color, hat

def draw_ball(hat):
    """Draw a ball using list element."""
    color = random_number.choice(hat)
    hat.remove(color)
    return color, hat

def new_hat():
    colors = 'black', 'red', 'blue'    # (tuple of strings)
    hat = []
    for color in colors:
        for i in range(4):
            hat.append(color)
    return hat

n = int(raw_input('How many balls are to be drawn? '))
N = int(raw_input('How many experiments? '))

```

```
# run experiments:
M = 0 # no of successes
for e in range(N):
    hat = new_hat()
    balls = [] # the n balls we draw
    for i in range(n):
        color, hat = draw_ball(hat)
        balls.append(color)
    if balls.count('black') >= 2: # at least two black balls?
        M += 1
print 'Probability:', float(M)/N
```

Running the program with  $n = 5$  (drawing 5 balls each time) and  $N = 4000$  gives a probability of 0.57. Drawing only 2 balls at a time reduces the probability to about 0.09.

One can with the aid of probability theory derive theoretical expressions for such probabilities, but it is much simpler to let the computer perform a large number of experiments to estimate an approximate probability.

A class version of the code in this section is better than the code presented, because we avoid shuffling the `hat` variable in and out of functions. Exercise 8.17 asks you to design and implement a class `Hat`.

### 8.3.4 Example: Policies for Limiting Population Growth

China has for many years officially allowed only one child per couple. However, the success of the policy has been somewhat limited. One challenge is the current overrepresentation of males in the population (families have favored sons to live up). An alternative policy is to allow each couple to continue getting children until they get a son. We can simulate both policies and see how a population will develop under the “one child” and the “one son” policies. Since we expect to work with a large population over several generations, we aim at vectorized code at once.

Suppose we have a collection of  $n$  individuals, called `parents`, consisting of males and females randomly drawn such that a certain portion (`male_portion`) constitutes males. The `parents` array holds integer values, 1 for male and 2 for females. We can introduce constants, `MALE=1` and `FEMALE=2`, to make the code easier to read. Our task is to see how the `parents` array develop from one generation to the next under the two policies. Let us first show how to draw the random integer array `parents` where there is a probability `male_portion` of getting the value `MALE`:

```
r = random.random(n)
parents = zeros(n, int)
MALE = 1; FEMALE = 2
parents[r < male_portion] = MALE
parents[r >= male_portion] = FEMALE
```

The number of potential couples is the minimum of males and females. However, only a fraction (`fertility`) of the couples will actually get a child. Under the perfect “one child” policy, these couples can have one child each:

```
males = len(parents[parents==MALE])
females = len(parents) - males
couples = min(males, females)
n = int(fertility*couples) # couples that get a child

# the next generation, one child per couple:
r = random.random(n)
children = zeros(n, int)
children[r < male_portion] = MALE
children[r >= male_portion] = FEMALE
```

The code for generating a new population will be needed in every generation. Therefore, it is natural to collect the last statements in a separate function such that we can repeat the statements when needed.

```
def get_children(n, male_portion, fertility):
    n = int(fertility*n)
    r = random.random(n)
    children = zeros(n, int)
    children[r < male_portion] = MALE
    children[r >= male_portion] = FEMALE
    return children
```

Under the “one son” policy, the families can continue getting a new child until they get the first son:

```
# first try:
children = get_children(couples, male_portion, fertility)

# continue with getting a new child for each daughter:
daughters = children[children == FEMALE]
while len(daughters) > 0:
    new_children = get_children(len(daughters),
                               male_portion, fertility)
    children = concatenate((children, new_children))
    daughters = new_children[new_children == FEMALE]
```

The program `birth_policy.py` organizes the code segments above for the two policies into a function `advance_generation`, which we can call repeatedly to see the evolution of the population.

```
def advance_generation(parents, policy='one child',
                      male_portion=0.5, fertility=1.0):
    males = len(parents[parents==MALE])
    females = len(parents) - males
    couples = min(males, females)

    if policy == 'one child':
        children = get_children(couples, male_portion, fertility)
    elif policy == 'one son':
        # first try at getting a child:
        children = get_children(couples, male_portion, fertility)
        # continue with getting a new child for each daughter:
```

```

    daughters = children[children == FEMALE]
    while len(daughters) > 0:
        new_children = get_children(len(daughters),
                                    male_portion, fertility)
        children = concatenate((children, new_children))
        daughters = new_children[new_children == FEMALE]
    return children

```

The simulation is then a matter of repeated calls to `advance_generation`:

```

N = 1000000          # population size
male_portion = 0.51
fertility = 0.92
# start with a "perfect" generation of parents:
parents = get_children(N, male_portion=0.5, fertility=1.0)
print 'one son policy, start: %d' % len(parents)
for i in range(10):
    parents = advance_generation(parents, 'one son',
                                male_portion, fertility)
    print '%3d: %d' % (i+1, len(parents))

```

Under ideal conditions with unit fertility and a `male_portion` of 0.5, the program predicts that the “one child” policy halves the population from one generation to the next, while the “one son” policy, where we expect each couple to get one daughter and one son on average, keeps the population constant. Increasing `male_portion` slightly and decreasing `fertility`, which corresponds more to reality, will in both cases lead to a reduction of the population. You can try the program out with various values of these input parameters.

An obvious extension is to incorporate the effect that a portion of the population does not follow the policy and get  $c$  children on average. The program `birth_policy.py` can account for the effect, which is quite dramatic: If 1% of the population does not follow the “one son” policy and get 4 children on average, the population grows with 50% over 10 generations (`male_portion` and `fertility` kept at the ideal values 0.5 and 1, respectively).

Normally, simple models like the difference equations (5.9) and (5.12), or the differential equations (B.11) or (B.23), are used to model population growth. However, these models track the number of individuals through time with a very simple growth factor from one generation to the next. The model above tracks each individual in the population and applies rules involving random actions to each individual. Such a detailed and much more computer-time consuming model can be used to see the effect of different policies. Using the results of this detailed model, we can (sometimes) estimate growth factors for simpler models so that these mimic the overall effect on the population size. Exercise 8.24 asks you to investigate if a certain realization of the “one son” policy leads to simple exponential growth.

## 8.4 Simple Games

This section presents the implementation of some simple games based on drawing random numbers. The games can be played by two humans, but here we consider a human versus the computer.

### 8.4.1 Guessing a Number

*The Game.* The computer determines a secret number, and the player shall guess the number. For each guess, the computer tells if the number is too high or too low.

*The Implementation.* We let the computer draw a random integer in an interval known to the player, let us say  $[1, 100]$ . In a `while` loop the program prompts the player for a guess, reads the guess, and checks if the guess is higher or lower than the drawn number. An appropriate message is written to the screen. We think the algorithm can be expressed directly as executable Python code:

```
import random as random_number
number = random_number.randint(1, 100)
attempts = 0 # count no of attempts to guess the number
guess = 0
while guess != number:
    guess = eval(raw_input('Guess a number: '))
    attempts += 1
    if guess == number:
        print 'Correct! You used', attempts, 'attempts!'
        break
    elif guess < number:
        print 'Go higher!'
    else:
        print 'Go lower!'
```

The program is available as the file `guessnumber.py`. Try it out! Can you come up with a strategy for reducing the number of attempts? See Exercise 8.25 for an automatic investigation of two possible strategies.

### 8.4.2 Rolling Two Dice

*The Game.* The player is supposed to roll two dice, and on beforehand guess the sum of the eyes. If the guess on the sum is  $n$  and it turns out to be right, the player earns  $n$  euros. Otherwise, the player must pay 1 euro. The machine plays in the same way, but the machine's guess of the number of eyes is a uniformly distributed number between 2 and 12. The player determines the number of rounds,  $r$ , to play, and receives  $r$  euros as initial capital. The winner is the one that has the largest amount of euros after  $r$  rounds, or the one that avoids to lose all the money.



*The Implementation.* There are three actions that we can naturally implement as functions: (i) roll two dice and compute the sum; (ii) ask the player to guess the number of eyes; (iii) draw the computer's guess of the number of eyes. One soon realizes that it is as easy to implement this game for an arbitrary number of dice as it is for two dice. Consequently we can introduce `ndice` as the number of dice. The three functions take the following forms:

```
import random as random_number

def roll_dice_and_compute_sum(ndice):
    return sum([random_number.randint(1, 6) \
                for i in range(ndice)])

def computer_guess(ndice):
    return random_number.randint(ndice, 6*ndice)

def player_guess(ndice):
    return input('Guess the sum of the no of eyes '\
                'in the next throw: ')
```

We can now implement one round in the game for the player or the computer. The round starts with a capital, a guess is performed by calling the right function for guessing, and the capital is updated:

```
def play_one_round(ndice, capital, guess_function):
    guess = guess_function(ndice)
    throw = roll_dice_and_compute_sum(ndice)
    if guess == throw:
        capital += guess
    else:
        capital -= 1
    return capital, throw, guess
```

Here, `guess_function` is either `computer_guess` or `player_guess`.

With the `play_one_round` function we can run a number of rounds involving both players:

```
def play(nrounds, ndice=2):
    # start capital:
    player_capital = computer_capital = nrounds

    for i in range(nrounds):
        player_capital, throw, guess = \
            play_one_round(ndice, player_capital, player_guess)
        print 'YOU guessed %d, got %d' % (guess, throw)
        if player_capital == 0:
            print 'Machine won!'; sys.exit(0)

        computer_capital, throw, guess = \
            play_one_round(ndice, computer_capital, computer_guess)

        print 'Machine guessed %d, got %d' % (guess, throw)
        if computer_capital == 0:
            print 'You won!'; sys.exit(0)

        print 'Status: you have %d euros, machine has %d euros' % \
            (player_capital, computer_capital)

    if computer_capital > player_capital:
```

```

        winner = 'Machine'
    else:
        winner = 'You'
    print winner, 'won!'

```

The name of the program is `ndice.py`.

*Example.* Here is a session (with a fixed seed of 20):

```

Guess the sum of the no of eyes in the next throw: 7
YOU guessed 7, got 11
Machine guessed 10, got 8
Status: you have 9 euros, machine has 9 euros

Guess the sum of the no of eyes in the next throw: 9
YOU guessed 9, got 10
Machine guessed 11, got 6
Status: you have 8 euros, machine has 8 euros

Guess the sum of the no of eyes in the next throw: 9
YOU guessed 9, got 9
Machine guessed 3, got 8
Status: you have 17 euros, machine has 7 euros

```

Exercise 8.27 asks you to perform simulations to determine whether a certain strategy can make the player win over the computer in the long run.

*A Class Version.* We can cast the previous code segment in a class. Many will argue that a class-based implementation is closer to the problem being modeled and hence easier to modify or extend.

A natural class is `Dice`, which can throw  $n$  dice:

```

class Dice:
    def __init__(self, n=1):
        self.n = n    # no of dice

    def throw(self):
        return [random_number.randint(1,6) \
                for i in range(self.n)]

```

Another natural class is `Player`, which can perform the actions of a player. Functions can then make use of `Player` to set up a game. A `Player` has a name, an initial capital, a set of dice, and a `Dice` object to throw the object:

```

class Player:
    def __init__(self, name, capital, guess_function, ndice):
        self.name = name
        self.capital = capital
        self.guess_function = guess_function
        self.dice = Dice(ndice)

    def play_one_round(self):
        self.guess = self.guess_function(self.dice.n)
        self.throw = sum(self.dice.throw())
        if self.guess == self.throw:
            self.capital += self.guess
        else:
            self.capital -= 1
        self.message()
        self.broke()

```

```
def message(self):
    print '%s guessed %d, got %d' % \
        (self.name, self.guess, self.throw)

def broke(self):
    if self.capital == 0:
        print '%s lost!' % self.name
        sys.exit(0) # end the program
```

The guesses of the computer and the player are specified by functions:

```
def computer_guess(ndice):
    # any of the outcomes (sum) is equally likely:
    return random_number.randint(ndice, 6*ndice)

def player_guess(ndice):
    return input('Guess the sum of the no of eyes '\
        'in the next throw: ')
```

The key function to play the whole game, utilizing the `Player` class for the computer and the user, can be expressed as

```
def play(nrounds, ndice=2):
    player = Player('YOU', nrounds, player_guess, ndice)
    computer = Player('Computer', nrounds, computer_guess, ndice)

    for i in range(nrounds):
        player.play_one_round()
        computer.play_one_round()
        print 'Status: user have %d euro, machine has %d euro\n' % \
            (player.capital, computer.capital)

    if computer.capital > player.capital:
        winner = 'Machine'
    else:
        winner = 'You'
    print winner, 'won!'
```

The complete code is found in the file `ndice2.py`. There is no new functionality compared to the `ndice.py` implementation, just a new and better structuring of the code.

## 8.5 Monte Carlo Integration

One of the earliest applications of random numbers was numerical computation of integrals, that is, a non-random (deterministic) problem. Here we shall address two related methods for computing  $\int_a^b f(x)dx$ .

### 8.5.1 Standard Monte Carlo Integration

Let  $x_1, \dots, x_n$  be uniformly distributed random numbers between  $a$  and  $b$ . Then

$$(b-a) \frac{1}{n} \sum_{i=1}^n f(x_i) \quad (8.7)$$

is an approximation to the integral  $\int_a^b f(x)dx$ . This method is usually referred to as *Monte Carlo integration*. It is easy to interpret (8.7). A well-known result from calculus is that the integral of a function  $f$  over  $[a, b]$  equals the mean value of  $f$  over  $[a, b]$  multiplied by the length of the interval,  $b - a$ . If we approximate the mean value of  $f(x)$  by the mean of  $n$  randomly distributed function evaluations  $f(x_i)$ , we get the method (8.7).

We can implement (8.7) in a small function:

```
import random as random_number

def MCint(f, a, b, n):
    s = 0
    for i in range(n):
        x = random_number.uniform(a, b)
        s += f(x)
    I = (float(b-a)/n)*s
    return I
```

One normally needs a large  $n$  to obtain good results with this method, so a faster vectorized version of the MCint function is handy:

```
from numpy import *

def MCint_vec(f, a, b, n):
    x = random.uniform(a, b, n)
    s = sum(f(x))
    I = (float(b-a)/n)*s
    return I
```

Let us try the Monte Carlo integration method on a simple linear function  $f(x) = 2 + 3x$ , integrated from 1 to 2. Most other numerical integration methods will integrate such a linear function exactly, regardless of the number of function evaluations. This is not the case with Monte Carlo integration. It would be interesting to see how the quality of the Monte Carlo approximation increases  $n$ . To plot the evolution of the integral approximation we must store intermediate  $I$  values. This requires a slightly modified MCint method:

```
def MCint2(f, a, b, n):
    s = 0
    # store the intermediate integral approximations in an
    # array I, where I[k-1] corresponds to k function evals.
    I = zeros(n)
    for k in range(1, n+1):
        x = random_number.uniform(a, b)
        s += f(x)
        I[k-1] = (float(b-a)/k)*s
    return I
```

Note that we let  $k$  go from 1 to  $n$  while the indices in  $I$ , as usual, go from 0 to  $n-1$ . Since  $n$  can be very large, the  $I$  array may consume more memory than what we have on the computer. Therefore, we decide to store only every  $N$  values of the approximation. Determining if a value is to be stored or not can then be computed by the mod function (see page 423 or Exercise 2.45):

```
for k in range(1, n+1):
    ...
    if k % N == 0:
        # store
```

That is, every time  $k$  can be divided by  $N$  without any remainder, we store the value. The complete function takes the following form:

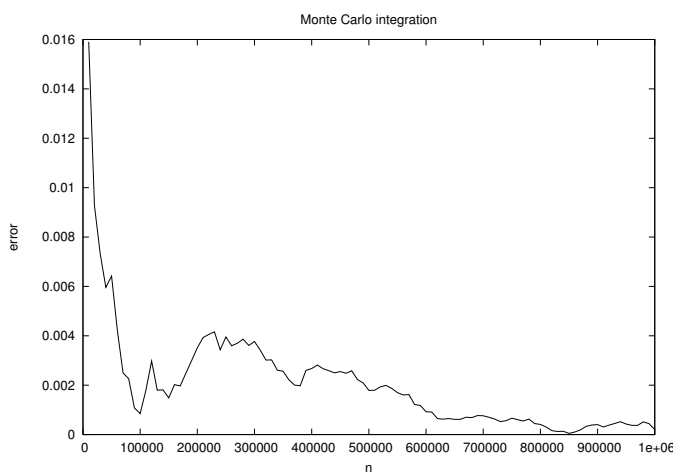
```
def MCInt3(f, a, b, n, N=100):
    s = 0
    # store every N intermediate integral approximations in an
    # array I and record the corresponding k value
    I_values = []
    k_values = []
    for k in range(1, n+1):
        x = random_number.uniform(a, b)
        s += f(x)
        if k % N == 0:
            I = (float(b-a)/k)*s
            I_values.append(I)
            k_values.append(k)
    return k_values, I_values
```

Now we have the tools to plot the error in the Monte Carlo approximation as a function of  $n$ :

```
def f1(x):
    return 2 + 3*x

k, I = MCInt3(f1, 1, 2, 1000000, N=10000)
from scitools.std import plot
error = 6.5 - array(I)
plot(k, error, title='Monte Carlo integration',
      xlabel='n', ylabel='error')
```

Figure 8.4 shows the resulting plot.



**Fig. 8.4** The convergence of Monte Carlo integration applied to  $\int_1^2 (2 + 3x)dx$ .

For functions of one variable, method (8.7) requires many points and is inefficient compared to other integration rules. Most integra-

tion rules have an error that reduces with increasing  $n$ , typically like  $n^{-r}$  for some  $r > 0$ . For the Trapezoidal rule,  $r = 2$ , while  $r = 1/2$  for Monte Carlo integration, which means that this method converges quite slowly compared to the Trapezoidal rule. However, for functions of many variables, Monte Carlo integration in high space dimension completely outperforms methods like the Trapezoidal rule and Simpson's rule. There are also many ways to improve the performance of (8.7), basically by being “smart” in drawing the random numbers (this is called variance reducing techniques).

### 8.5.2 Computing Areas by Throwing Random Points

Think of some geometric region  $G$  in the plane and a surrounding bounding box  $B$  with geometry  $[x_L, x_H] \times [y_L, y_H]$ . One way of computing the area of  $G$  is to draw  $N$  random points inside  $B$  and count how many of them,  $M$ , that lie inside  $G$ . The area of  $G$  is then the fraction  $M/N$  ( $G$ 's fraction of  $B$ 's area) times the area of  $B$ ,  $(x_H - x_L)(y_H - y_L)$ . Phrased differently, this method is a kind of dart game where you record how many hits there are inside  $G$  if every throw hits uniformly within  $B$ .

Let us formulate this method for computing the integral  $\int_a^b f(x)dx$ . The important observation is that this integral is the area under the curve  $y = f(x)$  and above the  $x$  axis, between  $x = a$  and  $x = b$ . We introduce a rectangle  $B$ ,

$$B = \{(x, y) \mid a \leq x \leq b, 0 \leq y \leq m\},$$

where  $m \leq \max_{x \in [a, b]} f(x)$ . The algorithm for computing the area under the curve is to draw  $N$  random points inside  $B$  and count how many of them,  $M$ , that are above the  $x$  axis and below the  $y = f(x)$  curve, see Figure 8.5. The area or integral is then estimated by

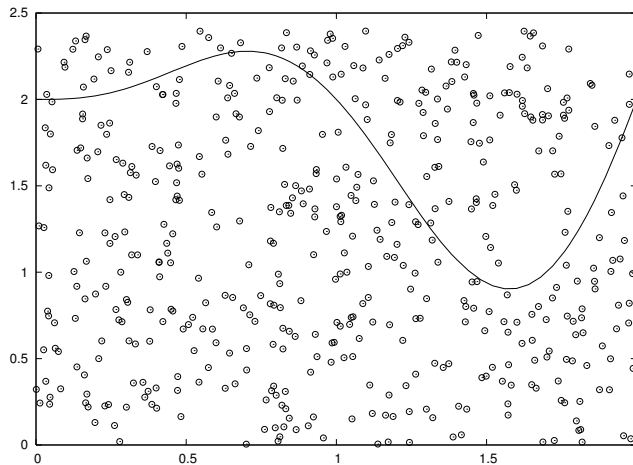
$$\frac{M}{N}m(b-a).$$

First we implement the “dart method” by a simple loop over points:

```
def MCint_area(f, a, b, n, m):
    below = 0 # counter for no of points below the curve
    for i in range(n):
        x = random_number.uniform(a, b)
        y = random_number.uniform(0, m)
        if y <= f(x):
            below += 1
    area = below/float(n)*m*(b-a)
    return area
```

Note that this method draws twice as many random numbers as the previous method.

A vectorized implementation reads



**Fig. 8.5** The “dart” method for computing integrals. When  $M$  out of  $N$  random points in the rectangle  $[0, 2] \times [0, 2.4]$  lie under the curve, the area under the curve is estimated as the  $M/N$  fraction of the area of the rectangle, i.e.,  $(M/N)2 \cdot 2.4$ .

```
def MCint_area_vec(f, a, b, n, m):
    x = random.uniform(a, b, n)
    y = random.uniform(0, m, n)
    below = y[y < f(x)].size
    area = below/float(n)*m*(b-a)
    return area
```

Even for 2 million random numbers the plain loop version is not that slow as it executes within some seconds on a slow laptop. Nevertheless, if you need the integration being repeated many times inside another calculation, the superior efficiency of the vectorized version may be important. We can quantify the efficiency gain by the aid of the timer `time.clock()` in the following way (see Appendix E.6.1):

```
import time
t0 = time.clock()
print MCint_area(f1, a, b, n, fmax)
t1 = time.clock() # time of MCint_area is t1-t0
print MCint_area_vec(f1, a, b, n, fmax)
t2 = time.clock() # time of MCint_area_vec is t2-t1
print 'loop/vectorized fraction:', (t1-t0)/(t2-t1)
```

With  $n = 10^6$  I achieved a factor of about 16 in favor of the vectorized version on an IBM laptop.

## 8.6 Random Walk in One Space Dimension

In this section we shall simulate a collection of particles that move around in a random fashion. This type of simulations are fundamental in physics, biology, chemistry as well as other sciences and can be used to describe many phenomena. Some application areas include molecular

motion, heat conduction, quantum mechanics, polymer chains, population genetics, brain research, hazard games, and pricing of financial instruments.

Imagine that we have some particles that perform random moves, either to the right or to the left. We may flip a coin to decide the movement of each particle, say head implies movement to the right and tail means movement to the left. Each move is one unit length. Physicists use the term *random walk* for this type of movement of a particle<sup>8</sup>.

The movement is also known as “drunkard’s walk”. You may have experienced this after a very wet night on a pub: you step forward and backward in a random fashion. Since these movements on average make you stand still, and since you know that you normally reach home within reasonable time, the model is not good for a real walk. We need to add a *drift* to the walk, so the probability is greater for going forward than backward. This is an easy adjustment, see Exercise 8.33. What may come as a surprise is the following fact: even when there is equal probability of going forward and backward, one can prove mathematically that the drunkard will always reach his home. Or more precisely, he will get home in finite time (“almost surely” as the mathematicians must add to this statement). Exercise 8.34 asks you to experiment with this fact. For many practical purposes, “finite time” does not help much as there might be more steps involved than the time it takes to get sufficiently sober to remove the completely random component of the walk.

### 8.6.1 Basic Implementation

How can we implement  $n_s$  random steps of  $n_p$  particles in a program? Let us introduce a coordinate system where all movements are along the  $x$  axis. An array of  $x$  values then holds the positions of all particles. We draw random numbers to simulate flipping a coin, say we draw from the integers 1 and 2, where 1 means head (movement to the right) and 2 means tail (movement to the left). We think the algorithm is conveniently expressed directly as a complete Python program:

```
import random as random_number
import numpy
np = 4                                # no of particles
ns = 100                             # no of steps
positions = numpy.zeros(np)           # all particles start at x=0
HEAD = 1; TAIL = 2                   # constants

for step in range(ns):
    for p in range(np):
        coin = random_number.randint(1,2) # flip coin
```

<sup>8</sup> You may try this yourself: flip the coin and make one step to the left or right, and repeat this process.



```

if coin == HEAD:
    positions[p] += 1    # one unit length to the right
elif coin == TAIL:
    positions[p] -= 1    # one unit length to the left

```

This program is found in the file `walk1D.py`.

### 8.6.2 Visualization

We may add some visualization of the movements by inserting a `plot` command at the end of the `step` loop and a little pause to better separate the frames in the animation<sup>9</sup>:

```

plot(positions, y, 'ko3', axis=[xmin, xmax, -0.2, 0.2])
time.sleep(0.2) # pause

```

Recall from Chapter 4 that in an animation like this the axis must be kept fixed. We know that in  $n_s$  steps, no particle can move longer than  $n_s$  unit lengths to the right or to the left so the extent of the  $x$  axis becomes  $[-n_s, n_s]$ . However, the probability of reaching these lower or upper limit is very small<sup>10</sup>. Most of the movements will take place in the center of the plot. We may therefore shrink the extent of the axis to better view the movements. It is known that the expected extent of the particles is of the order  $\sqrt{n_s}$ , so we may take the maximum and minimum values in the plot as  $\pm 2\sqrt{n_s}$ . However, if a position of a particle exceeds these values, we extend `xmax` and `xmin` by  $2\sqrt{n_s}$  in positive and negative  $x$  direction, respectively.

The  $y$  positions of the particles are taken as zero, but it is necessary to have some extent of the  $y$  axis, otherwise the coordinate system collapses and most plotting packages will refuse to draw the plot. Here we have just chosen the  $y$  axis to go from -0.2 to 0.2. You can find the complete program in `src/random/walk1Dp.py`. The `np` and `ns` parameters can be set as the first two command-line arguments:

---

Terminal

---

```
walk1Dp.py 6 200
```

---

It is hard to claim that this program has astonishing graphics. In Chapter 8.7, where we let the particles move in two space dimensions, the graphics gets much more exciting.

### 8.6.3 Random Walk as a Difference Equation

The random walk process can easily be expressed in terms of a difference equation (Chapter 5). Let  $x_n$  be the position of the particle at

<sup>9</sup> These actions require `from scitools.std import *` and `import time`.

<sup>10</sup> The probability is  $2^{-n_s}$ , which becomes about  $10^{-9}$  for 30 steps.

time  $n$ . This position is an evolution from time  $n - 1$ , obtained by adding a random variable  $s$  to the previous position  $x_{n-1}$ , where  $s = 1$  has probability  $1/2$  and  $s = -1$  has probability  $1/2$ . In statistics, the expression “probability of event A” is written  $P(A)$ . We can therefore write  $P(s = 1) = 1/2$  and  $P(s = -1) = 1/2$ . The difference equation can now be expressed mathematically as

$$x_n = x_{n-1} + s, \quad x_0 = 0, \quad P(s = 1) = P(s = -1) = 1/2. \quad (8.8)$$

This equation governs the motion of one particle. For a collection  $m$  of particles we introduce  $x_n^{(i)}$  as the position of the  $i$ -th particle at the  $n$ -th time step. Each  $x_n^{(i)}$  is governed by (8.8), and all the  $s$  values in each of the  $m$  difference equations are independent of each other.

#### 8.6.4 Computing Statistics of the Particle Positions

Scientists interested in random walks are in general not interested in the graphics of our `walk1D.py` program, but more in the statistics of the positions of the particles at each step. We may therefore, at each step, compute a histogram of the distribution of the particles along the  $x$  axis, plus estimate the mean position and the standard deviation. These mathematical operations are easily accomplished by letting the SciTools function `compute_histogram` and the numpy functions `mean` and `std` operate on the `positions` array (see Chapter 8.1.5)<sup>11</sup> :

```
mean_pos = mean(positions)
stdev_pos = std(positions)
pos, freq = compute_histogram(positions, nbins=int(xmax),
                               piecewise_constant=True)
```

We can plot the particles as circles, as before, and add the histogram and vertical lines for the mean and the positive and negative standard deviation (the latter indicates the “width” of the distribution of particles). The vertical lines can be defined by the six lists

```
xmean, ymean = [mean_pos, mean_pos], [yminv, ymaxv]
xstdv1, ystdv1 = [stdev_pos, stdev_pos], [yminv, ymaxv]
xstdv2, ystdv2 = [-stdev_pos, -stdev_pos], [yminv, ymaxv]
```

where `yminv` and `ymaxv` are the minimum and maximum  $y$  values of the vertical lines. The following command plots the position of every particle as circles, the histogram as a curve, and the vertical lines with a thicker line:

<sup>11</sup> The number of bins in the histogram is just based on the extent of the particles. It could also have been a fixed number.

```

plot(positions, y, 'ko3',      # particles as circles
     pos, freq, 'r',          # histogram
     xmean, ymean, 'r2',      # mean position as thick line
     xstdv1, ystdv1, 'b2',    # +1 standard dev.
     xstdv2, ystdv2, 'b2',    # -1 standard dev.
     axis=[xmin, xmax, ymin, ymax],
     title='random walk of %d particles after %d steps' % \
           (np, step+1))

```

This plot is then created at every step in the random walk. By observing the graphics, one will soon realize that the computation of the extent of the  $y$  axis in the plot needs some considerations. We have found it convenient to base  $y_{\max}$  on the maximum value of the histogram ( $\max(\text{freq})$ ), plus some space (chosen as 10 percent of  $\max(\text{freq})$ ). However, we do not change the  $y_{\max}$  value unless it is more than 0.1 different from the previous  $y_{\max}$  value (otherwise the axis “jumps” too often). The minimum value,  $y_{\min}$ , is set to  $y_{\min} = -0.1 * y_{\max}$  every time we change the  $y_{\max}$  value. The complete code is found in the file `walk1Ds.py`. If you try out 2000 particles and 30 steps, the final graphics becomes like that in Figure 8.6. As the number of steps is increased, the particles are dispersed in the positive and negative  $x$  direction, and the histogram gets flatter and flatter. Letting  $\hat{H}(i)$  be the histogram value in interval number  $i$ , and each interval having width  $\Delta x$ , the probability of finding a particle in interval  $i$  is  $\hat{H}(i)\Delta x$ . It can be shown mathematically that the histogram is an approximation to the probability density function of the normal distribution (1.6) (see page 45), with mean zero and standard deviation  $s \sim \sqrt{n}$ , where  $n$  is the step number.

### 8.6.5 Vectorized Implementation

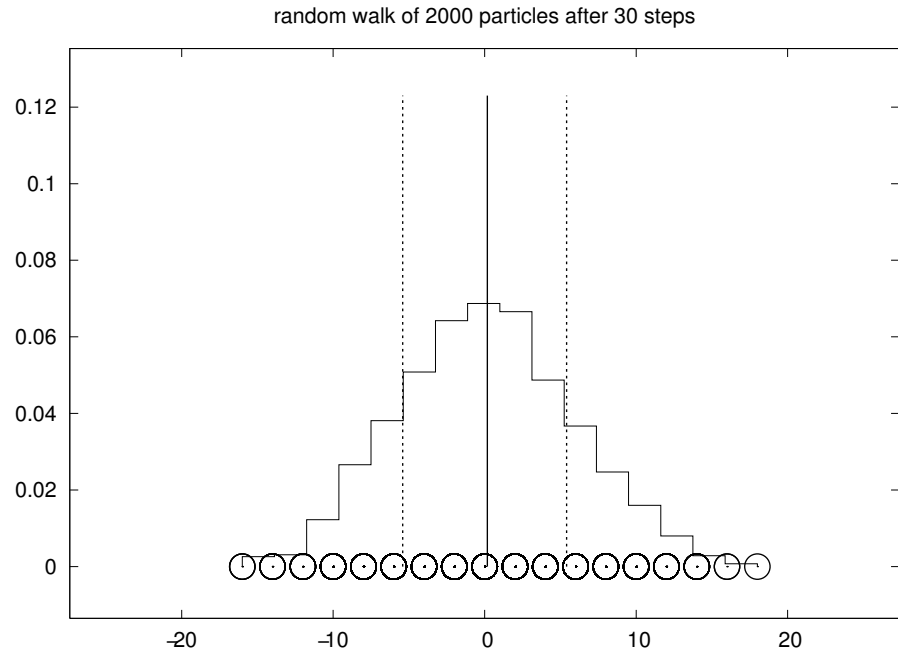
There is no problem with the speed of our one-dimensional random walkers in the `walk1Dp.py` or `walk1Ds.py` programs, but in real-life applications of such simulation models, we often have a very large number of particles performing a very large number of steps. It is then important to make the implementation as efficient as possible. Two loops over all particles and all steps, as we have in the programs above, become very slow compared to a vectorized implementation.

A vectorized implementation of a one-dimensional walk should utilize the functions `randint` or `random_integers` from `numpy.random`. A first idea may be to draw steps for all particles at a step simultaneously. Then we repeat this process in a loop from 0 to  $n_s - 1$ . However, these repetitions are just new vectors of random numbers, and we may avoid the loop if we draw  $n_p \times n_s$  random numbers at once:

```

moves = random.randint(1, 3, size=np*ns)
# or
moves = random.random_integers(1, 2, size=np*ns)

```



**Fig. 8.6** Particle positions (circles), histogram (piecewise constant curve), and vertical lines indicating the mean value and the standard deviation from the mean after a one-dimensional random walk of 2000 particles for 30 steps.

The values are now either 1 or 2, but we want  $-1$  or  $1$ . A simple scaling and translation of the numbers transform the 1 and 2 values to  $-1$  and  $1$  values:

```
moves = 2*moves - 3
```

Then we can create a two-dimensional array out of `moves` such that `moves[i,j]` is the  $i$ -th step of particle number  $j$ :

```
moves.shape = (ns, np)
```

It does not make sense to plot the evolution of the particles and the histogram in the vectorized version of the code, because the point with vectorization is to speed up the calculations, and the visualization takes much more time than drawing random numbers, even in the `walk1Dp.py` and `walk1Ds.py` programs from Chapter 8.6.4. We therefore just compute the positions of the particles inside a loop over the steps and some simple statistics. At the end, after  $n_s$  steps, we plot the histogram of the particle distribution along with circles for the positions of the particles. The rest of the program, found in the file `walk1Dv.py`, looks as follows:

```

positions = zeros(np)
for step in range(ns):
    positions += moves[step, :]

    mean_pos, stdev_pos = mean(positions), std(positions)
    print mean_pos, stdev_pos

nbins = int(3*sqrt(ns))    # no of intervals in histogram
pos, freq = compute_histogram(positions, nbins,
                              piecewise_constant=True)

plot(positions, zeros(np), 'ko3',
      pos, freq, 'r',
      axis=[min(positions), max(positions), -0.05, 1.1*max(freq)],
      hardcopy='tmp.ps')

```

## 8.7 Random Walk in Two Space Dimensions

A random walk in two dimensions performs a step either to the north, south, west, or east, each one with probability  $1/4$ . To demonstrate this process, we introduce  $x$  and  $y$  coordinates of  $n_p$  particles and draw random numbers among 1, 2, 3, or 4 to determine the move. The positions of the particles can easily be visualized as small circles in an  $xy$  coordinate system.

### 8.7.1 Basic Implementation

The algorithm described above is conveniently expressed directly as a complete working program:

```

def random_walk_2D(np, ns, plot_step):
    xpositions = zeros(np)
    ypositions = zeros(np)
    # extent of the axis in the plot:
    ymax = 3*sqrt(ns); xmin = -ymax

    NORTH = 1; SOUTH = 2; WEST = 3; EAST = 4 # constants

    for step in range(ns):
        for i in range(np):
            direction = random_number.randint(1, 4)
            if direction == NORTH:
                ypositions[i] += 1
            elif direction == SOUTH:
                ypositions[i] -= 1
            elif direction == EAST:
                xpositions[i] += 1
            elif direction == WEST:
                xpositions[i] -= 1

        # plot just every plot_step steps:
        if (step+1) % plot_step == 0:
            plot(xpositions, ypositions, 'ko',
                 axis=[xmin, ymax, xmin, ymax],
                 title='%d particles after %d steps' % \

```

```

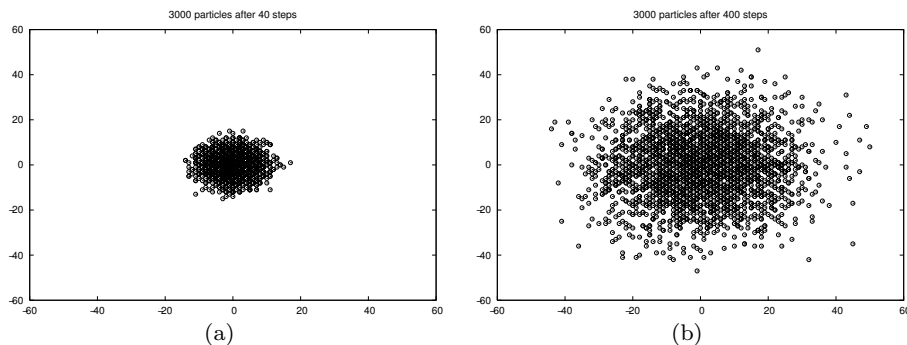
        (np, step+1),
        hardcopy='tmp_%03d.eps' % (step+1))
    return xpositions, ypositions

# main program:
import random as random_number
random_number.seed(10)
import sys
from scitools.std import zeros, plot, sqrt

np      = int(sys.argv[1]) # number of particles
ns      = int(sys.argv[2]) # number of steps
plot_step = int(sys.argv[3]) # plot every plot_step steps
x, y = random_walk_2D(np, ns, plot_step)

```

The program is found in the file `walk2D.py`. Figure 8.7 shows two snapshots of the distribution of 3000 particles after 40 and 400 steps. These plots were generated with command-line arguments `3000 400 20`, the latter implying that we visualize the particles every 20 time steps only.



**Fig. 8.7** Location of 3000 particles starting at the origin and performing a random walk: (a) 40 steps; (b) 400 steps.

To get a feeling for the two-dimensional random walk you can try out only 30 particles for 400 steps and let each step be visualized (i.e., command-line arguments `30 400 1`). The update of the movements is now fast.

The `walk2D.py` program dumps the plots to PostScript files with names of the form `tmp_xxx.eps`, where `xxx` is the step number. We can create a movie out of these individual files using the `movie` function (Chapter 4.3.7) or the program `convert` from the ImageMagick suite<sup>12</sup>:

```
convert -delay 50 -loop 1000 tmp_*.eps movie.gif
```

All the plots are now put after each other as frames in a movie, with a delay of 50 ms between each frame. The movie will run in a loop 1000 times. Alternatively, we can create the movie with the `movie` function from Easyviz, inside a program:

<sup>12</sup> If you want to run this command from an IPython session, prefix `convert` with an exclamation mark: `!convert`.

```
from scitools.std import movie
movie('tmp_*.eps', encoder='convert', output_file='movie.gif')
```

The resulting movie file is named `movie.gif`, which can be viewed by the `animate` program (also from the ImageMagick program suite), just write `animate movie.gif`. Making and showing the movie are slow processes if a large number of steps are included in the movie – 100 steps or fewer are appropriate, but this depends on the power of your computer.

### 8.7.2 Vectorized Implementation

The `walk2D.py` program is quite slow. Now the visualization is much faster than the movement of the particles. Vectorization may speed up the `walk2D.py` program significantly. As in the one-dimensional phase, we draw all the movements at once and then invoke a loop over the steps to update the  $x$  and  $y$  coordinates. We draw  $n_s \times n_p$  numbers among 1, 2, 3, and 4. We then reshape the vector of random numbers to a two-dimensional array `moves[i, j]`, where  $i$  counts the steps,  $j$  counts the particles. The `if` test on whether the current move is to the north, south, east, or west can be vectorized using the `where` function (see Chapter 4.4.1). For example, if the random numbers for all particles in the current step are accessible in an array `this_move`, we could update the  $x$  positions by

```
xpositions += where(this_move == EAST, 1, 0)
xpositions -= where(this_move == WEST, 1, 0)
```

provided `EAST` and `WEST` are constants, equal to 3 and 4, respectively. A similar construction can be used for the  $y$  moves.

The complete program is listed below:

```
def random_walk_2D(np, ns, plot_step):
    xpositions = zeros(np)
    ypositions = zeros(np)
    moves = random.random_integers(1, 4, size=ns*np)
    moves.shape = (ns, np)

    # estimate max and min positions:
    xymin = 3*sqrt(ns); xymin = -xymin

    NORTH = 1; SOUTH = 2; WEST = 3; EAST = 4 # constants

    for step in range(ns):
        this_move = moves[step,:]
        ypositions += where(this_move == NORTH, 1, 0)
        ypositions -= where(this_move == SOUTH, 1, 0)
        xpositions += where(this_move == EAST, 1, 0)
        xpositions -= where(this_move == WEST, 1, 0)

    # just plot every plot_step steps:
    if (step+1) % plot_step == 0:
        plot(xpositions, ypositions, 'ko',
            axis=[xymin, xymin, xymin, xymin],
```

```

        title='%d particles after %d steps' % \
            (np, step+1),
        hardcopy='tmp_%03d.eps' % (step+1))
    return xpositions, ypositions

# main program:
from scitools.std import *
random.seed(11)

np = int(sys.argv[1]) # number of particles
ns = int(sys.argv[2]) # number of steps
plot_step = int(sys.argv[3]) # plot each plot_step step
x, y = random_walk_2D(np, ns, plot_step)

```

You will easily experience that this program, found in the file `walk2Dv.py`, runs significantly faster than the `walk2D.py` program.

## 8.8 Summary

### 8.8.1 Chapter Topics

*Drawing Random Numbers.* Random numbers can be scattered throughout an interval in various ways, specified by the *distribution* of the numbers. We have considered a uniform distribution (Chapter 8.1.2) and a normal (or Gaussian) distribution (Chapter 8.1.6). Table 8.1 shows the syntax for generating random numbers of these two distributions, using either the standard scalar `random` module in Python or the vectorized `numpy.random` module.

**Table 8.1** Summary of important functions for drawing random numbers. `N` is the array length in vectorized drawing, while `m` and `s` represent the mean and standard deviation values of a normal distribution.

	<code>random</code>	<code>numpy.random</code>
uniform numbers in $[0, 1)$	<code>random()</code>	<code>random(N)</code>
uniform numbers in $[a, b)$	<code>uniform(a, b)</code>	<code>uniform(a, b, N)</code>
integers in $[a, b]$	<code>randint(a, b)</code>	<code>randint(a, b+1, N)</code>
		<code>random_integers(a, b, N)</code>
Gaussian numbers, mean $m$ , st.dev. $s$	<code>gauss(m, s)</code>	<code>normal(m, s, N)</code>
set seed ( $i$ )	<code>seed(i)</code>	<code>seed(i)</code>
shuffle list $a$ (in-place)	<code>shuffle(a)</code>	<code>shuffle(a)</code>
choose a random element in list $a$	<code>choice(a)</code>	

*Typical Probability Computation.* Many programs performing probability computations draw a large number `N` of random numbers and count how many times `M` a random number leads to some true condition (Monte Carlo simulation):



```
import random as random_number
M = 0
for i in xrange(N):
    r = random_number.randint(a, b)
    if condition:
        M += 1
print 'Probability estimate:', float(M)/N
```

For example, if we seek the probability that we get at least four eyes when throwing a dice, we choose the random number to be the number of eyes, i.e., an integer in the interval  $[1, 6]$  ( $a=1$ ,  $b=6$ ) and `condition` becomes `r >= 4`.

For large  $N$  we can speed up such programs by vectorization, i.e., drawing all random numbers at once in a big array and use operations on the array to find  $M$ . The similar vectorized version of the program above looks like

```
from numpy import *
r = random.uniform(a, b, N)
M = sum(condition)
# or
M = sum(where(condition, 1, 0))
print 'Probability estimate:', float(M)/N
```

(Combinations of boolean expressions in the `condition` argument to `where` requires special constructs as outlined in Exercise 8.14.) Make sure you use `sum` from `numpy`, when operating on large arrays, and not the much slower built-in `sum` function in Python.

*Statistical Measures.* Given an array of random numbers, the following code computes the mean, variance, and standard deviation of the numbers and finally displays a plot of the histogram, which reflects how the numbers are statistically distributed:

```
from scitools.std import mean, var, std, compute_histogram
m = mean(numbers)
v = var(numbers)
s = std(numbers)
x, y = compute_histogram(numbers, 50, piecewise_constant=True)
plot(x, y)
```

### 8.8.2 Summarizing Example: Random Growth

Chapter 5.1.1 contains mathematical models for how an investment grows when there is an interest rate being added to the investment at certain intervals. The model can easily allow for a time-varying interest rate, but for forecasting the growth of an investment, it is difficult to predict the future interest rate. One commonly used method is to build a probabilistic model for the development of the interest rate, where the rate is chosen randomly at random times. This gives a random

growth of the investment, but by simulating many random scenarios we can compute the mean growth and use the standard deviation as a measure of the uncertainty of the predictions.

*Problem.* Let  $p$  be the annual interest rate in a bank in percent. Suppose the interest is added to the investment  $q$  times per year. The new value of the investment,  $x_n$ , is given by the previous value of the investment,  $x_{n-1}$ , plus the  $p/q$  percent interest:

$$x_n = x_{n-1} + \frac{p}{100q} x_{n-1}.$$

Normally, the interest is added daily ( $q = 360$  and  $n$  counts days), but for efficiency in the computations later we shall assume that the interest is added monthly, so  $q = 12$  and  $n$  counts months.

The basic assumption now is that  $p$  is random and varies with time. Suppose  $p$  increases with a random amount  $\gamma$  from one month to the next:

$$p_n = p_{n-1} + \gamma.$$

A typical size of  $p$  adjustments is 0.5. However, the central bank does not adjust the interest every month. Instead this happens every  $M$  months on average. The probability of a  $\gamma \neq 0$  can therefore be taken as  $1/M$ . In a month where  $\gamma \neq 0$ , we may say that  $\gamma = m$  with probability  $1/2$  or  $\gamma = -m$  with probability  $1/2$  if it is equally likely that the rate goes up as down (this is not a good assumption, but a more complicated evolution of  $\gamma$  is postponed now).

*Solution.* First we must develop the precise formulas to be implemented. The difference equations for  $x_n$  and  $p_n$  are in simple in the present case, but the details computing  $\gamma$  must be worked out. In a program, we can draw two random numbers to estimate  $\gamma$ : one for deciding if  $\gamma \neq 0$  and the other for determining the sign of the change. Since the probability for  $\gamma \neq 0$  is  $1/M$ , we can draw a number  $r_1$  among the integers  $1, \dots, M$  and if  $r_1 = 1$  we continue with drawing a second number  $r_2$  among the integers 1 and 2. If  $r_2 = 1$  we set  $\gamma = m$ , and if  $r_2 = 2$  we set  $\gamma = -m$ . We must also assure that  $p_n$  does not take on unreasonable values, so we choose  $p_n < 1$  and  $p_n > 15$  as cases where  $p_n$  is not changed.

The mathematical model for the investment must track both  $x_n$  and  $p_n$ . Below we express with precise mathematics the equations for  $x_n$  and  $p_n$  and the computation of the random  $\gamma$  quantity:

$$x_n = x_{n-1} + \frac{p_{n-1}}{12 \cdot 100} x_{n-1}, \quad i = 1, \dots, N \quad (8.9)$$

$$r_1 = \text{random integer in } [1, M] \quad (8.10)$$

$$r_2 = \text{random integer in } [1, 2] \quad (8.11)$$

$$\gamma = \begin{cases} m, & \text{if } r_1 = 1 \text{ and } r_2 = 1, \\ -m, & \text{if } r_1 = 1 \text{ and } r_2 = 2, \\ 0, & \text{if } r_1 \neq 1 \end{cases} \quad (8.12)$$

$$p_n = p_{n-1} + \begin{cases} \gamma, & \text{if } p_{n-1} + \gamma \in [1, 15], \\ 0, & \text{otherwise} \end{cases} \quad (8.13)$$

We remark that the evolution of  $p_n$  is much like a random walk process (Chapter 8.6), the only differences is that the plus/minus steps are taken at some random points among the times  $0, 1, 2, \dots, N$  rather than at all times  $0, 1, 2, \dots, N$ . The random walk for  $p_n$  also has barriers at  $p = 1$  and  $p = 15$ , but that is common in a standard random walk too.

Each time we calculate the  $x_n$  sequence in the present application, we get a different development because of the random numbers involved. We say that one development of  $x_0, \dots, x_n$  is a *path* (or realization, but since the realization can be viewed as a curve  $x_n$  or  $p_n$  versus  $n$  in this case, it is common to use the word path). Our Monte Carlo simulation approach consists of computing a large number of paths, as well as the sum of the path and the sum of the paths squared. From the latter two sums we can compute the mean and standard deviation of the paths to see the average development of the investment and the uncertainty of this development. Since we are interested in complete paths, we need to store the complete sequence of  $x_n$  for each path. We may also be interested in the statistics of the interest rate so we store the complete sequence  $p_n$  too.

Programs should be built in pieces so that we can test each piece before testing the whole program. In the present case, a natural piece is a function that computes one path of  $x_n$  and  $p_n$  with  $N$  steps, given  $M$ ,  $m$ , and the initial conditions  $x_0$  and  $p_0$ . We can then test this function before moving on to calling the function a large number of times. An appropriate code may be

```
def simulate_one_path(N, x0, p0, M, m):
    x = zeros(N+1)
    p = zeros(N+1)
    index_set = range(0, N+1)

    x[0] = x0
    p[0] = p0

    for n in index_set[1:]:
        x[n] = x[n-1] + p[n-1]/(100.0*12)*x[n-1]

        # update interest rate p:
        r = random_number.randint(1, M)
        if r == 1:
```

```

        # adjust gamma:
        r = random_number.randint(1, 2)
        gamma = m if r == 1 else -m
    else:
        gamma = 0
    pn = p[n-1] + gamma
    p[n] = pn if 1 <= pn <= 15 else p[n-1]
    return x, p

```

Testing such a function is challenging because the result is different each time because of the random numbers. A first step in verifying the implementation is to turn off the randomness ( $m = 0$ ) and check that the deterministic parts of the difference equations are correctly computed:

```

x, p = simulate_one_path(3, 1, 10, 1, 0)
print x

```

The output becomes

```
[ 1.          1.00833333  1.01673611  1.02520891]
```

These numbers can quickly be checked against a formula of the type (5.4) on page 237 in an interactive session:

```

>>> def g(x0, n, p):
...     return x0*(1 + p/(12.*100))**n
...
>>> g(1, 1, 10)
1.0083333333333333
>>> g(1, 2, 10)
1.0167361111111111
>>> g(1, 3, 10)
1.0252089120370369

```

We can conclude that our function works well when there is no randomness. A next step is to carefully examine the code that computes `gamma` and compare with the mathematical formulas.

Simulating many paths and computing the average development of  $x_n$  and  $p_n$  is a matter of calling `simulate_one_path` repeatedly, use two arrays `xm` and `pm` to collect the sum of `x` and `p`, respectively, and finally obtain the average path by dividing `xm` and `pm` by the number of paths we have computed:

```

def simulate_n_paths(n, N, L, p0, M, m):
    xm = zeros(N+1)
    pm = zeros(N+1)
    for i in range(n):
        x, p = simulate_one_path(N, L, p0, M, m)
        # accumulate paths:
        xm += x
        pm += p
    # compute average:
    xm /= float(n)
    pm /= float(n)
    return xm, pm

```

We can also compute the standard deviation of the paths using formulas (8.3) and (8.6), with  $x_j$  as either an `x` or a `p` array. It might

happen that small round-off errors generate a small *negative* variance, which mathematically should have been slightly greater than zero. Taking the square root will then generate complex arrays and problems with plotting. To avoid this problem, we therefore replace all negative elements by zeros in the variance arrays before taking the square root. The new lines for computing the standard deviation arrays `xs` and `ps` are indicated below:

```
def simulate_n_paths(n, N, x0, p0, M, m):
    ...
    xs = zeros(N+1) # standard deviation of x
    ps = zeros(N+1) # standard deviation of p
    for i in range(n):
        x, p = simulate_one_path(N, x0, p0, M, m)
        # accumulate paths:
        xm += x
        pm += p
        xs += x**2
        ps += p**2

    ...
    # compute standard deviation:
    xs = xs/float(n) - xm*xm # variance
    ps = ps/float(n) - pm*pm # variance
    # remove small negative numbers (round off errors):
    xs[xs < 0] = 0
    ps[ps < 0] = 0
    xs = sqrt(xs)
    ps = sqrt(ps)
    return xm, xs, pm, ps
```

A remark regarding the efficiency of array operations is appropriate here. The statement `xs += x**2` could equally well, from a mathematical point of view, be written as `xs = xs + x**2`. However, in this latter statement, two extra arrays are created (one for the squaring and one for the sum), while in the former only one array (`x**2`) is made. Since the paths can be long and we make many simulations, such optimizations can be important.

One may wonder whether `x**2` is “smart” in the sense that squaring is detected and computed as `x*x`, not as a general (slow) power function. This is indeed the case for arrays, as we have investigated in the little test program `smart_power.py` in the `random` directory. This program applies time measurement methods from Appendix E.6.2.

Our `simulate_n_paths` function generates four arrays which are natural to visualize. Having a mean and a standard deviation curve, it is often common to plot the mean curve with one color or linetype and then two curves, corresponding to plus one and minus one standard deviation, with another less visible color. This gives an indication of the mean development and the uncertainty of the underlying process. We therefore make two plots: one with `xm`, `xm+xs`, and `xm-xs`, and one with `pm`, `pm+ps`, and `pm-ps`.

Both for debugging and curiosity it is handy to have some plots of a few actual paths. We may pick out 5 paths from the simulations and visualize these:

```
def simulate_n_paths(n, N, x0, p0, M, m):
    ...
    for i in range(n):
        ...
        # show 5 random sample paths:
        if i % (n/5) == 0:
            figure(1)
            plot(x, title='sample paths of investment')
            hold('on')
            figure(2)
            plot(p, title='sample paths of interest rate')
            hold('on')
        figure(1); hardcopy('tmp_sample_paths_investment.eps')
        figure(2); hardcopy('tmp_sample_paths_interestrate.eps')
    ...
    return ...
```

Note the use of `figure`: we need to hold on both figures to add new plots and switch between the figures, both for plotting and making the final hardcopy.

After the visualization of sample paths we make the mean  $\pm$  standard deviation plots by this code:

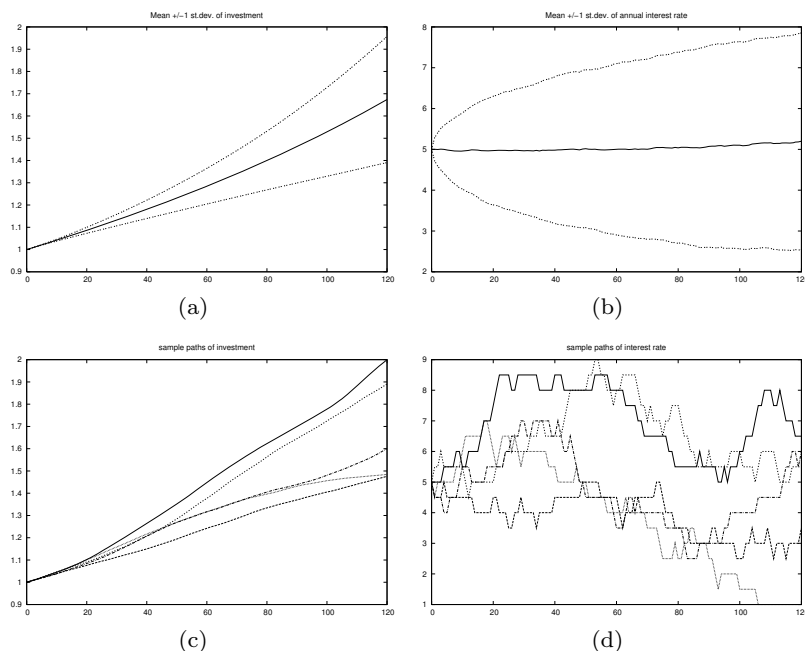
```
xm, xs, pm, ps = simulate_n_paths(n, N, x0, p0, M, m)
figure(3)
months = range(len(xm)) # indices along the x axis
plot(months, xm, 'r',
      months, xm-xs, 'y',
      months, xm+xs, 'y',
      title='Mean +/- 1 st.dev. of investment',
      hardcopy='tmp_mean_investment.eps')
figure(4)
plot(months, pm, 'r',
      months, pm-ps, 'y',
      months, pm+ps, 'y',
      title='Mean +/- 1 st.dev. of annual interest rate',
      hardcopy='tmp_mean_interestrate.eps')
```

The complete program for simulating the investment development is found in the file `growth_random.py`.

Running the program with the input data

```
x0 = 1          # initial investment
p0 = 5          # initial interest rate
N = 10*12       # number of months
M = 3           # p changes (on average) every M months
n = 1000        # number of simulations
m = 0.5         # adjustment of p
```

and initializing the seed of the random generator to 1, we get four plots, which are shown in Figure 8.8.



**Fig. 8.8** Development of an investment with random jumps of the interest rate at random points of time: (a) mean value of investment  $\pm$  one standard deviation; (b) mean value of the interest rate  $\pm$  one standard deviation; (c) five paths of the investment development; (d) five paths of the interest rate development.

## 8.9 Exercises

### Exercise 8.1. *Flip a coin $N$ times.*

Make a program that simulates flipping a coin  $N$  times. Print out “tail” or “head” for each flip and let the program count the number of heads. (Hint: Use `r = random.random()` and define head as `r <= 0.5` or draw an integer among  $\{1, 2\}$  with `r = random.randint(1, 2)` and define head when `r` is 1.) Name of program file: `flip_coin.py`.  $\diamond$

### Exercise 8.2. *Compute a probability.*

What is the probability of getting a number between 0.5 and 0.6 when drawing uniformly distributed random numbers from the interval  $[0, 1)$ ? To answer this question empirically, let a program draw  $N$  such random numbers using Python’s standard `random` module, count how many of them,  $M$ , that fall in the interval  $(0.5, 0.6)$ , and compute the probability as  $M/N$ . Run the program with the four values  $N = 10^i$  for  $i = 1, 2, 3, 6$ . Name of program file: `compute_prob.py`.  $\diamond$

### Exercise 8.3. *Choose random colors.*

Suppose we have eight different colors. Make a program that chooses one of these colors at random and writes out the color. Hint: Use a list of color names and use the `choice` function in the `random` module to pick a list element. Name of program file: `choose_color.py`.  $\diamond$

**Exercise 8.4.** *Draw balls from a hat.*

Suppose there are 40 balls in a hat, of which 10 are red, 10 are blue, 10 are yellow, and 10 are purple. What is the probability of getting two blue and two purple balls when drawing 10 balls at random from the hat? Name of program file: `4balls_from10.py`. ◇

**Exercise 8.5.** *Probabilities of rolling dice.*

1. You throw a die. What is the probability of getting a 6?
2. You throw a die four times in a row. What is the probability of getting 6 all the times?
3. Suppose you have thrown the die three times with 6 coming up all times. What is the probability of getting a 6 in the fourth throw?
4. Suppose you have thrown the die 100 times and experienced a 6 in every throw. What do you think about the probability of getting a 6 in the next throw?

First try to solve the questions from a theoretical or common sense point of view. Thereafter, make functions for simulating cases 1, 2, and 3. Name of program file: `rolling_dice.py`. ◇

**Exercise 8.6.** *Estimate the probability in a dice game.*

Make a program for estimating the probability of getting at least one 6 when throwing  $n$  dice. Read  $n$  and the number of experiments from the command line. (To verify the program, you can compare the estimated probability with the exact result  $11/36$  when  $n = 2$ .) Name of program file: `one6_2dice.py`. ◇

**Exercise 8.7.** *Decide if a dice game is fair.*

Somebody suggests the following game. You pay 1 unit of money and are allowed to throw four dice. If the sum of the eyes on the dice is less than 9, you win 10 units of money, otherwise you lose your investment. Should you play this game? Answer the question by making a program that simulates the game. Name of program file: `sum9_4dice.py`. ◇

**Exercise 8.8.** *Adjust the game in Exer. 8.7.*

It turns out that the game in Exercise 8.7 is not fair, since you lose money in the long run. The purpose of this exercise is to adjust the winning award so that the game becomes fair, i.e., that you neither lose nor win money in the long run.

Make a program that computes the probability  $p$  of getting a sum less than  $s$  when rolling  $n$  dice. Name of program file: `sum_s_ndice_fair.py`.

If the cost of each game is  $q$  units of money, the game is fair if the payment in case you win is  $r = q/p$ . Run the program you made for  $s = 9$  and  $n = 4$ , which corresponds to the game in Exercise 8.7, and compute the corresponding  $p$ . Modify the program from Exercise 8.7 so that the award is  $r = 1/p$ , and run that program to see that now



the game is fair, i.e., you neither win nor lose money in a large number of games.

*Explanation.* The formula for a fair game can be developed as follows. Let  $p = M/N$  be the probability of winning, which means that you in the long run win  $M$  out of  $N$  games. The cost is  $Nq$  and the income is  $Mr$ . To make the net income  $Mr - Nq$  zero, which is the requirement of a fair game, we get  $r = qN/M = q/p$ . (This reasoning is based on common sense and an intuitive interpretation of probability. More precise reasoning from probability theory will introduce the game as an experiment with two outcomes, either you win with probability  $p$  and or lose with probability  $1-p$ . The expected payment is then the sum of probabilities times the corresponding net incomes:  $-q(1-p) - (r-q)p$  (recall that the net income in a winning game is  $r-q$ ). A fair game has zero expected payment, i.e.,  $r = q/p$ .)  $\diamond$

**Exercise 8.9.** *Probabilities of throwing two dice.*

Make a computer program for throwing two dice a large number of times. Record the sum of the eyes each time and count how many times each of the possibilities for the sum (2, 3, ..., 12) appear. A dictionary with the sum as key and count as value is convenient here. Divide the counts by the total number of trials such that you get the frequency of each possible sum. Write out the frequencies and compare them with exact probabilities. (To find the exact probabilities, set up all the  $6 \times 6$  possible outcomes of throwing two dice, and then count how many of them that has a sum  $s$  for  $s = 2, 3, \dots, 12$ .) Name of program file: `freq_2dice.py`.  $\diamond$

**Exercise 8.10.** *Compute the probability of drawing balls.*

A hat has 20 balls, 5 red, 5 yellow, 5 green, and 5 brown. We draw  $n \geq 3$  balls at random. What is the probability of getting

- at least one red and one brown ball?
- exactly one red ball?
- exactly two red balls?
- at least three green balls?

Use Monte Carlo simulation to compute the probabilities and write out the answers to the four questions for  $n = 3, 5, 7, 10, 15$ . Name of program file: `draw_balls.py`.  $\diamond$

**Exercise 8.11.** *Compute the probability of hands of cards.*

Use the `Deck.py` module (in `src/random`) and the `same_rank` and `same_suit` functions from the `cards` module to compute the following probabilities by Monte Carlo simulation:

- exactly two pairs among five cards,
- four or five cards of the same suit among five cards,
- four-of-a-kind among five cards.

Name of program file: `card_hands.py`. ◇

**Exercise 8.12.** *Play with vectorized boolean expressions.*

Using the `numpy.random` module, make an array containing  $N$  uniformly distributed random numbers between 0 and 1. Print out the arrays `r <= 0.5`, `r[r <= 0.5]`, `where(r <= 0.5, 1, 0)` and convince yourself that you understand what these arrays express. We want to compute how many of the elements in `r` that are less than or equal to 0.5. How can this be done in a vectorized way, i.e., without explicit loops in the program, but solely with operations on complete arrays? Name of program file: `bool_vec.py`. ◇

**Exercise 8.13.** *Vectorize the program from Exer. 8.1.*

Simulate flipping a coin  $N$  times and write out the number of tails. The code should be vectorized, i.e., there must be no loops in Python. Hint: Use ideas from Exercise 8.12. Name of program file: `flip_coin_vec.py`. ◇

**Exercise 8.14.** *Vectorize the code in Exer. 8.2.*

The purpose of this exercise is to speed up the code in Exercise 8.2 by vectorization. Hint: First draw an array `r` with a large number of random numbers in  $[0, 1)$ . The simplest way to count how many elements in `r` that lie between 0.5 and 0.6, is to first extract the elements larger than 0.5: `r1 = r[r>0.5]`, and then extract the elements in `r1` that are less than 0.6 and get the size of this array: `r1[r1<=0.6].size`. Name of program file: `compute_prob_vec.py`.

*Remark.* An alternative and more complicated method is to use the `where` function. The condition (the first argument to `where`) is now a compound boolean expression `0.5 <= r <= 0.6`, but this cannot be used with NumPy arrays. Instead one must test for `0.5 <= r` and `r <= 0.6`. The needed boolean construction in the `where` call is `operator.and_(0.5 <= r, r <= 0.6)`. See also the discussion of the same topic in Chapter 4.4.1. ◇

**Exercise 8.15.** *Throw dice and compute a small probability.*

Compute the probability of getting 6 eyes on all dice when rolling 7 dice. Since you need a large number of experiments in this case (see the first paragraph of Chapter 8.3), you can save quite some simulation time by using a vectorized implementation. Name of program file: `roll_7dice.py`. ◇

**Exercise 8.16.** *Difference equation for random numbers.*

Simple random number generators are based on simulating difference equations. Here is a typical set of two equations:

$$x_n = (ax_{n-1} + c) \bmod m, \quad (8.14)$$

$$y_n = x_n/m, \quad (8.15)$$

for  $n = 1, 2, \dots$ . A seed  $x_0$  must be given to start the sequence. The numbers  $y_1, y_2, \dots$ , represent the random numbers and  $x_0, x_1, \dots$  are “help” numbers. Although  $y_n$  is completely deterministic from (8.14)–(8.15), the sequence  $y_n$  *appears* random. The mathematical expression  $p \bmod q$  is coded as `p % q` in Python.

Use  $a = 8121$ ,  $c = 28411$ , and  $m = 134456$ . Solve the system (8.14)–(8.15) in a function that generates and returns  $N$  random numbers. Make a histogram to examine the distribution of the numbers (the  $y_n$  numbers are randomly distributed if the histogram is approximately flat). Name of program file: `diffeq_random.py`.  $\diamond$

**Exercise 8.17.** *Make a class for drawing balls from a hat.*

Consider the example about drawing colored balls from a hat in Chapter 8.3.3. It could be handy to have an object that acts as a hat:

```
# make a hat with balls of 3 colors, each color appearing
# on 4 balls:
hat = Hat(colors=('red', 'black', 'blue'), number_of_each_color=4)

# draw 3 balls at random
balls = hat.draw(number_of_balls=3)
```

Realize such code with a class `Hat`. You can borrow useful code from the `balls_in_hat.py` program and ideas from Chapter 8.2.5. Use the `Hat` class to solve the probability problem from Exercise 8.4. Name of program file: `Hat.py`.  $\diamond$

**Exercise 8.18.** *Independent vs. dependent random numbers.*

Generate a sequence of  $N$  independent random variables with values 0 or 1 and print out this sequence without space between the numbers (i.e., as 001011010110111010).

The next task is to generate random zeros and ones that are dependent. If the last generated number was 0, the probability of generating a new 0 is  $p$  and a new 1 is  $1 - p$ . Conversely, if the last generated was 1, the probability of generating a new 1 is  $p$  and a new 0 is  $1 - p$ . Since the new value depends on the last one, we say the variables are dependent. Implement this algorithm in a function returning an array of  $N$  zeros and ones. Print out this array in the condense format as described above.

Choose  $N = 80$  and try  $p = 0.5$ ,  $0 = 0.8$  and  $p = 0.9$ . Can you describe the differences between sequences of independent and dependent random variables? Name of program file: `dependent_random_variables.py`.  $\diamond$

**Exercise 8.19.** *Compute the probability of flipping a coin.*

Modify the program from either Exercise 8.1 or 8.13 to incorporate the following extensions: look at a subset  $N_1 \leq N$  of the experiments and compute probability of getting a head ( $M_1/N_1$ , where  $M_1$  is the number of heads in  $N_1$  experiments). Choose  $N = 1000$  and print out

the probability for  $N_1 = 10, 100, 500, 1000$ . (Generate just  $N$  numbers once in the program.) How do you think the accuracy of the computed probability vary with  $N_1$ ? Is the output compatible with this expectation? Name of program file: `flip_coin_prob.py`. ◇

**Exercise 8.20.** *Extend Exer. 8.19.*

We address the same problem as in Exercise 8.19, but now we want to study the probability of getting a head,  $p$ , as a function of  $N_1$ , i.e., for  $N_1 = 1, \dots, N$ . We also want to vectorize all operations in the code. A first try to compute the probability array for  $p$  is

```
h = where(r <= 0.5, 1, 0)
p = zeros(N)
for i in range(N):
    p[i] = sum(h[:i+1])/float(i+1)
```

An array  $q[i] = \text{sum}(h[:i])$  reflects a *cumulative sum* and can be efficiently generated by the `cumsum` function in `numpy`: `q = cumsum(h)`. Thereafter we can compute  $p$  by  $q/I$ , where  $I[i]=i+1$  and  $I$  can be computed by `arange(1,N+1)` or `r_[1:N+1]`. Implement both the loop over  $i$  and the vectorized version based on `cumsum` and check in the program that the resulting  $p$  array has the same elements (for this purpose you have to compare `float` elements and you can use the `float_eq` function from `SciTools`, see Exercise 2.51, or the `allclose` function in `numpy` (`float_eq` actually uses `allclose` for array arguments)). Plot  $p$  against  $I$  for the case where  $N = 10000$ . Annotate the axis and the plot with relevant text. Name of program file: `flip_coin_prob_developm.py`. ◇

**Exercise 8.21.** *Simulate the problems in Exer. 3.26.*

Exercise 3.26 describes some problems that can be solved exactly using the formula (3.8), but we can also simulate these problems and find approximate numbers for the probabilities. That is the task of this exercise.

Make a general function `simulate_binomial(p, n, x)` for running  $n$  experiments, where each experiment have two outcomes, with probabilities  $p$  and  $1-p$ . The  $n$  experiments constitute a “success” if the outcome with probability  $p$  occurs exactly  $x$  times. The `simulate_binomial` function must repeat the  $n$  experiments  $N$  times. If  $M$  is the number of “successes” in the  $N$  experiments, the probability estimate is  $M/N$ . Let the function return this probability estimate together with the error (the exact result is (3.8)). Simulate the three cases in Exercise 3.26 using this function. Name of program file: `simulate_binomial_problems.py`. ◇

**Exercise 8.22.** *Simulate a poker game.*

Make a program for simulating the development of a poker (or simplified poker) game among  $n$  players. Use ideas from Chapter 8.2.4. Name of program file: `poker.py`. ◇

**Exercise 8.23.** *Write a non-vectorized version of a code.*

Read the file `birth_policy.py` containing the code from Chapter 8.3.4. To prove that you understand what is going on in this simulation, replace all the vectorized code by explicit loops over the random arrays. For such code it is natural to use Python's standard `random` module instead of `numpy.random`. However, to verify your alternative implementation it is convenient to have the same sequence of random numbers in the two programs. Therefore, use `numpy`'s `random` module, but use it like the standard Python `random` module, i.e., draw real numbers one at a time instead of a whole array at once. Name of program file: `birth_policy2.py`.  $\diamond$

**Exercise 8.24.** *Estimate growth in a simulation model.*

The simulation model in Chapter 8.3.4 predicts the number of individuals from generation to generation. Make a simulation of the “one son” policy with 10 generations, a male portion of 0.51 among newborn babies, set the fertility to 0.92, and assume that 6% of the population will break the law and want 6 children in a family. These parameters implies a significant growth of the population. See if you can find a factor  $r$  such that the number of individuals in generation  $n$  fulfills the difference equation

$$x_n = (1 + r)x_{n-1}.$$

Hint: Compute  $r$  for two consecutive generations  $x_{n-1}$  and  $x_n$  ( $r = x_n/x_{n-1} - 1$ ) and see if  $r$  is approximately constant through the evolution of the generations. Name of program file: `growth_birth_policy.py`.

$\diamond$

**Exercise 8.25.** *Investigate guessing strategies for Ch. 8.4.1.*

In the game from Chapter 8.4.1 it is smart to use the feedback from the program to track an interval  $[p, q]$  that must contain the secret number. Start with  $p = 1$  and  $q = 100$ . If the user guesses at some number  $n$ , update  $p$  to  $n + 1$  if  $n$  is less than the secret number (no need to care about numbers smaller than  $n + 1$ ), or update  $q$  to  $n - 1$  if  $n$  is larger than the secret number (no need to care about numbers larger than  $n - 1$ ).

Are there any smart strategies to pick a new guess  $s \in [p, q]$ ? To answer this question, investigate two possible strategies:  $s$  as the midpoint in the interval  $[p, q]$ , or  $s$  as a uniformly distributed random integer in  $[p, q]$ . Make a program that implements both strategies, i.e., the player is not prompted for a guess but the computer computes the guess based on the chosen strategy. Let the program run a large number of games and see if either of the strategies can be considered as superior in the long run. Name of program file: `strategies4guess.py`.  $\diamond$

**Exercise 8.26.** *Make a vectorized solution to Exer. 8.7.*

Vectorize the simulation program from Exercise 8.7 with the aid of the module `numpy.random` and the `numpy.sum` function. Name of program file: `sum9_4dice_vec.py`. ◇

**Exercise 8.27.** *Compare two playing strategies.*

Suggest a player strategy for the game in Chapter 8.4.2. Remove the question in the `player_guess` function in the file `src/random/ndice2.py`, and implement the chosen strategy instead. Let the program play a large number of games, and record the number of times the computer wins. Which strategy is best in the long run: the computer's or yours? Name of program file: `simulate_strategies1.py`. ◇

**Exercise 8.28.** *Solve Exercise 8.27 with different no. of dice.*

Solve Exercise 8.27 for two other cases with 3 and 50 dice, respectively. Name of program file: `simulate_strategies2.py`. ◇

**Exercise 8.29.** *Extend Exercise 8.28.*

Extend the program from Exercise 8.28 such that the computer and the player can use a different number of dice. Let the computer choose a random number of dice between 2 and 20. Experiment to find out if there is a favorable number of dice for the player. Name of program file: `simulate_strategies3.py`. ◇

**Exercise 8.30.** *Compute  $\pi$  by a Monte Carlo method.*

Use the method in Chapter 8.5.2 to compute  $\pi$  by computing the area of a circle. Choose  $G$  as the circle with its center at the origin and with unit radius, and choose  $B$  as the rectangle  $[-1, 1] \times [-1, 1]$ . A point  $(x, y)$  lies within  $G$  if  $x^2 + y^2 < 1$ . Compare the approximate  $\pi$  with `math.pi`. Name of program file: `MC_pi.py`. ◇

**Exercise 8.31.** *Do a variant of Exer. 8.30.*

This exercise has the same purpose of computing  $\pi$  as in Exercise 8.30, but this time you should choose  $G$  as a circle with center at  $(2, 1)$  and radius 4. Select an appropriate rectangle  $B$ . A point  $(x, y)$  lies within a circle with center at  $(x_c, y_c)$  and with radius  $R$  if  $(x - x_c)^2 + (y - y_c)^2 < R^2$ . Name of program file: `MC_pi2.py`. ◇

**Exercise 8.32.** *Compute  $\pi$  by a random sum.*

Let  $x_0, \dots, x_N$  be  $N + 1$  uniformly distributed random numbers between 0 and 1. Explain why the random sum  $S_N = \sum_{i=0}^N 2(1 - x_i^2)^{-1}$  is an approximation to  $\pi$ . (Hint: Interpret the sum as Monte Carlo integration and compute the corresponding integral exactly by hand.) Make a program for plotting  $S_N$  versus  $N$  for  $N = 10^k$ ,  $k = 0, 1/2, 1, 3/2, 2, 5/2, \dots, 6$ . Write out the difference between  $S_{10^6}$  and `pi` from the `math` module. Name of program file: `MC_pi_plot.py`. ◇

**Exercise 8.33.** *1D random walk with drift.*

Modify the `walk1D.py` program such that the probability of going to the right is  $r$  and the probability of going to the left is  $1 - r$  (draw numbers in  $[0, 1)$  rather than integers in  $\{1, 2\}$ ). Compute the average position of  $n_p$  particles after 100 steps, where  $n_p$  is read from the command line. Mathematically one can show that the average position approaches  $rn_s - (1 - r)n_s$  as  $n_p \rightarrow \infty$ . Write out this exact result together with the computed mean position with a finite number of particles. Name of program file: `walk1D_drift.py`.  $\diamond$

**Exercise 8.34.** *1D random walk until a point is hit.*

Set `np=1` in the `walk1Dv.py` program and modify the program to measure how many steps it takes for one particle to reach a given point  $x = x_p$ . Give  $x_p$  on the command line. Report results for  $x_p = 5, 50, 5000, 50000$ . Name of program file: `walk1Dv_hit_point.py`.  $\diamond$

**Exercise 8.35.** *Make a class for 2D random walk.*

The purpose of this exercise is to reimplement the `walk2D.py` program from Chapter 8.7.1 with the aid of classes. Make a class `Particle` with the coordinates  $(x, y)$  and the time step number of a particle as attributes. A method `move` moves the particle in one of the four directions and updates the  $(x, y)$  coordinates. Another class, `Particles`, holds a list of `Particle` objects and a `plotstep` parameter (as in `walk2D.py`). A method `move` moves all the particles one step, a method `plot` can make a plot of all particles, while a method `moves` performs a loop over time steps and calls `move` and `plot` in each step.

Equip the `Particle` and `Particles` classes with print functionality such that one can print out all particles in a nice way by saying `print p` (for a `Particles` instance `p`) or `print self` (inside a method). Hint: In `__str__`, apply the `pformat` function from the `pprint` module to the list of particles, and make sure that `__repr__` just reuse `__str__` in both classes.

To verify the implementation, print the first three positions of four particles in the `walk2D.py` program and compare with the corresponding results produced by the class-based implementation (the seed of the random number generator must of course be fixed identically in the two programs). You can just perform `p.move()` and `print p` three times in a `verify` function to do this verification task.

Organize the complete code as a module such that the classes `Particle` and `Particles` can be reused in other programs. The test block should call a `run(N)` method to run the walk for `N` steps, where `N` is given on the command line.

Compare the efficiency of the class version against the vectorized version in `walk2Dv.py`, using the techniques of Appendix E.6.1. Name of program file: `walk2Dc.py`.  $\diamond$

**Exercise 8.36.** *Vectorize the class code from Exer. 8.35.*

The program developed in Exercise 8.35 cannot be vectorized as long as we base the implementation on class `Particle`. However, if we remove that class and focus on class `Particles`, the latter can employ arrays for holding the positions of all particles and vectorized updates of these positions in the `moves` method. Use ideas from the `walk2Dv.py` program to vectorize class `Particle`. Verify the code against `walk2Dv.py` as explained in Exercise 8.35, and measure the efficiency gain over the version with class `Particle`. Name of program file: `walk2Dcv.py`. ◇

**Exercise 8.37.** *2D random walk with walls; scalar version.*

Modify the `walk2D.py` program or the `walk2Dc.py` program from Exercise 8.35 so that the walkers cannot walk outside a rectangular area  $A = [x_L, x_H] \times [y_L, y_H]$ . Do not move the particle if the new position of a particle is outside  $A$ . Name of program file: `walk2D_barrier.py`. ◇

**Exercise 8.38.** *2D random walk with walls; vectorized version.*

Modify the `walk2Dv.py` program so that the walkers cannot walk outside a rectangular area  $A = [x_L, x_H] \times [y_L, y_H]$ . Hint: First perform the moves of one direction. Then test if new positions are outside  $A$ . Such a test returns a boolean array that can be used as index in the position arrays to pick out the indices of the particles that have moved outside  $A$ . With this array index, one can move all particles outside  $A$  back to the relevant boundary of  $A$ . Name of program file: `walk2Dv_barrier.py`. ◇

**Exercise 8.39.** *Simulate the mixture of gas molecules.*

Suppose we have a box with a wall dividing the box into two equally sized parts. In one part we have a gas where the molecules are uniformly distributed in a random fashion. At  $t = 0$  we remove the wall. The gas molecules will now move around and eventually fill the whole box.

This physical process can be simulated by a 2D random walk inside a fixed area  $A$  as introduced in Exercises 8.37 and 8.38 (in reality the motion is three-dimensional, but we only simulate the two-dimensional part of it since we already have programs for doing this). Use the program from either Exercises 8.37 or 8.38 to simulate the process for  $A = [0, 1] \times [0, 1]$ . Initially, place 10000 particles at uniformly distributed random positions in  $[0, 1/2] \times [0, 1]$ . Then start the random walk and visualize what happens. Simulate for a long time and make a hardcopy of the animation (an animated GIF file, for instance). Is the end result what you would expect? Name of program file: `disorder1.py`.

Molecules tend to move randomly because of collisions and forces between molecules. We do not model collisions between particles in the random walk, but the nature of this walk, with random movements, simulates the effect of collisions. Therefore, the random walk can be used to model molecular motion in many simple cases. In particular, the random walk can be used to investigate how a quite ordered system,



where one gas fills one half of a box, evolves through time to a more disordered system.  $\diamond$

**Exercise 8.40.** *Simulate the mixture of gas molecules.*

Solve Exercise 8.39 when the wall dividing the box is not completely removed, but instead we make a small hole in the wall initially. Name of program file: `disorder2.py`.  $\diamond$

**Exercise 8.41.** *Guess beer brands.*

You are presented  $n$  glasses of beer, each containing a different brand. You are informed that there are  $m \geq n$  possible brands in total, and the names of all brands are given. For each glass, you can pay  $p$  euros to taste the beer, and if you guess the right brand, you get  $q \geq p$  euros back. Suppose you have done this before and experienced that you typically manage to guess the right brand  $T$  times out of 100, so that your probability of guessing the right brand is  $b = T/100$ .

Make a function `simulate(m, n, p, q, b)` for simulating the beer tasting process. Let the function return the amount of money earned and how many correct guesses ( $\leq n$ ) you made. Call `simulate` a large number of times and compute the average earnings and the probability of getting full score in the case  $m = n = 4$ ,  $p = 3$ ,  $q = 6$ , and  $b = 1/m$  (i.e., four glasses with four brands, completely random guessing, and a payback of twice as much as the cost). How much more can you earn from this game if your ability to guess the right brand is better, say  $b = 1/2$ ? Name of program file: `simulate_beer_tasting.py`.  $\diamond$

**Exercise 8.42.** *Simulate stock prices.*

A common mathematical model for the evolution of stock prices can be formulated as a difference equation

$$x_n = x_{n-1} + \Delta t \mu x_{n-1} + \sigma x_{n-1} \sqrt{\Delta t} r_{n-1}, \quad (8.16)$$

where  $x_n$  is the stock price at time  $t_n$ ,  $\Delta t$  is the time interval between two time levels ( $\Delta t = t_n - t_{n-1}$ ),  $\mu$  is the growth rate of the stock price,  $\sigma$  is the volatility of the stock price, and  $r_0, \dots, r_{n-1}$  are normally distributed random numbers with mean zero and unit standard deviation. An initial stock price  $x_0$  must be prescribed together with the input data  $\mu$ ,  $\sigma$ , and  $\Delta t$ .

We can make a remark that Equation (8.16) is a Forward Euler discretization of a stochastic differential equation for  $x(t)$ :

$$\frac{dx}{dt} = \mu x + \sigma N(t),$$

where  $N(t)$  is a so-called white noise random time series signal. Such equations play a central role in modeling of stock prices.

Make  $R$  realizations of (8.16) for  $n = 0, \dots, N$  for  $N = 5000$  steps over a time period of  $T = 180$  days with a step size  $\Delta t = T/N$ . Name of program file: `stock_prices.py`.  $\diamond$

**Exercise 8.43.** *Compute with option prices in finance.*

In this exercise we are going to consider the pricing of so-called Asian options. An Asian option is a financial contract where the owner earns money when certain market conditions are satisfied.

The contract is specified by a *strike price*  $K$  and a maturity time  $T$ . It is written on the average price of the underlying stock, and if this average is bigger than the strike  $K$ , the owner of the option will earn the difference. If, on the other hand, the average becomes less, the owner receives nothing, and the option matures in the value zero. The average is calculated from the last trading price of the stock for each day.

From the theory of options in finance, the price of the Asian option will be the expected present value of the payoff. We assume the stock price dynamics given as,

$$S(t+1) = (1+r)S(t) + \sigma S(t)\epsilon(t), \quad (8.17)$$

where  $r$  is the interest-rate, and  $\sigma$  is the volatility of the stock price. The time  $t$  is supposed to be measured in days,  $t = 0, 1, 2, \dots$ , while  $\epsilon(t)$  are independent identically distributed normal random variables with mean zero and unit standard deviation. To find the option price, we must calculate the expectation

$$p = (1+r)^{-T} \mathbb{E} \left[ \max \left( \frac{1}{T} \sum_{t=1}^T S(t) - K, 0 \right) \right]. \quad (8.18)$$

The price is thus given as the expected discounted payoff. We will use Monte Carlo simulations to estimate the expectation. Typically,  $r$  and  $\sigma$  can be set to  $r = 0.0002$  and  $\sigma = 0.015$ . Assume further  $S(0) = 100$ .

- a) Make a function that simulates a path of  $S(t)$ , that is, the function computes  $S(t)$  for  $t = 1, \dots, T$  for a given  $T$  based on the recursive definition in (8.17). The function should return the path as an array.
- b) Create a function that finds the average of  $S(t)$  from  $t = 1$  to  $t = T$ . Make another function that calculates the price of the Asian option based on  $N$  simulated averages. You may choose  $T = 100$  days and  $K = 102$ .
- c) Plot the price  $p$  as a function of  $N$ . You may start with  $N = 1000$ .
- d) Plot the error in the price estimation as a function  $N$  (assume that the  $p$  value corresponding to the largest  $N$  value is the “right” price). Try to fit a curve of the form  $c/\sqrt{N}$  for some  $c$  to this error plot. The purpose is to show that the error is reduced as  $1/\sqrt{N}$ .

Name of program file: `option_price.py`.

If you wonder where the values for  $r$  and  $\sigma$  come from, you will find the explanation in the following. A reasonable level for the yearly interest-rate is around 5%, which corresponds to a daily rate  $0.05/250 = 0.0002$ . The number 250 is chosen because a stock exchange is on average open this amount of days for trading. The value for  $\sigma$  is calculated as the volatility of the stock price, corresponding to the standard deviation of the daily returns of the stock defined as  $(S(t+1) - S(t))/S(t)$ . “Normally”, the volatility is around 1.5% a day. Finally, there are theoretical reasons why we assume that the stock price dynamics is driven by  $r$ , meaning that we consider the *risk-neutral* dynamics of the stock price when pricing options. There is an exciting theory explaining the appearance of  $r$  in the dynamics of the stock price. If we want to simulate a stock price dynamics mimicing what we see in the market,  $r$  in Equation (8.17) must be substituted with  $\mu$ , the expected return of the stock. Usually,  $\mu$  is higher than  $r$ .  $\diamond$

**Exercise 8.44.** *Compute velocity and acceleration.*

In a laboratory experiment waves are generated through the impact of a model slide into a wave tank. (The intention of the experiment is to model a future tsunami event in a fjord, generated by loose rocks that fall into the fjord.) At a certain location, the elevation of the surface, denoted by  $\eta$ , is measured at discrete points in time using an ultra-sound wave gauge. The result is a time series of vertical positions of the water surface elevations in meter:  $\eta(t_0), \eta(t_1), \eta(t_2), \dots, \eta(t_n)$ . There are 300 observations per second, meaning that the time difference between to neighboring measurement values  $\eta(t_i)$  and  $\eta(t_{i+1})$  is  $h = 1/300$  second.

Write a Python program that accomplishes the following tasks:

1. Read  $h$  from the command line.
2. Read the  $\eta$  values in the file `src/random/gauge.dat` into an array `eta`.
3. Plot `eta` versus the time values.
4. Compute the velocity  $v$  of the surface by the formula

$$v_i \approx \frac{\eta_{i+1} - \eta_{i-1}}{2h}, \quad i = 1, \dots, n-1.$$

Plot  $v$  versus time values in a separate plot.

5. Compute the acceleration  $a$  of the surface by the formula

$$a_i \approx \frac{\eta_{i+1} - 2\eta_i + \eta_{i-1}}{h^2}, \quad i = 1, \dots, n-1.$$

Plot  $a$  versus the time values in a separate plot.

Name of program file: `labstunami1.py`.

$\diamond$

**Exercise 8.45.** *Numerical differentiation of noisy signals.*

The purpose of this exercise is to look into numerical differentiation of time series signals that contain measurement errors. This insight might be helpful when analyzing the noise in real data from a laboratory experiment in Exercises 8.44 and 8.46.

1. Compute a signal

$$\bar{\eta}_i = A \sin\left(\frac{2\pi}{T} t_i\right), \quad t_i = i \frac{T}{40}, \quad i = 0, \dots, 200.$$

Display  $\bar{\eta}_i$  versus time  $t_i$  in a plot. Choose  $A = 1$  and  $T = 2\pi$ . Store the  $\bar{\eta}$  values in an array `etabar`.

2. Compute a signal with random noise  $E_i$ ,

$$\eta_i = \bar{\eta}_i + E_i,$$

$E_i$  is drawn from the normal distribution with mean zero and standard deviation  $\sigma = 0.04A$ . Plot this  $\eta_i$  signal as circles in the same plot as  $\bar{\eta}_i$ . Store the  $E_i$  in an array `E` for later use.

3. Compute the first derivative of  $\bar{\eta}_i$  by the formula

$$\frac{\bar{\eta}_{i+1} - \bar{\eta}_{i-1}}{2h}, \quad i = 1, \dots, n-1,$$

and store the values in an array `detabar`. Display the graph.

4. Compute the first derivative of the error term by the formula

$$\frac{E_{i+1} - E_{i-1}}{2h}, \quad i = 1, \dots, n-1,$$

and store the values in an array `dE`. Calculate the mean and the standard deviation of `dE`.

5. Plot `detabar` and `detabar + dE`. Use the result of the standard deviation calculations to explain the qualitative features of the graphs.
6. The second derivative of a time signal  $\eta_i$  can be computed by

$$\frac{\eta_{i+1} - 2\eta_i + \eta_{i-1}}{h^2}, \quad i = 1, \dots, n-1.$$

Use this formula on the `etabar` data and save the result in `d2etabar`. Also apply the formula to the `E` data and save the result in `d2E`. Plot `d2etabar` and `d2etabar + d2E`. Compute the standard deviation of `d2E` and compare with the standard deviation of `dE` and `E`. Discuss the plot in light of these standard deviations.

Name of program file: `sine_noise.py`.

◇

**Exercise 8.46.** *Model the noise in the data in Exer. 8.44.*

We assume that the measured data can be modeled as a smooth time signal  $\bar{\eta}(t)$  plus a random variation  $E(t)$ . Computing the velocity of  $\eta = \bar{\eta} + E$  results in a smooth velocity from the  $\bar{\eta}$  term and a noisy

signal from the  $E$  term. We can estimate the level of noise in the first derivative of  $E$  as follows. The random numbers  $E(t_i)$  are assumed to be independent and normally distributed with mean zero and standard deviation  $\sigma$ . It can then be shown that

$$\frac{E_{i+1} - E_{i-1}}{2h}$$

produces numbers that come from a normal distribution with mean zero and standard deviation  $2^{-1/2}h^{-1}\sigma$ . How much is the original noise, reflected by  $\sigma$ , magnified when we use this numerical approximation of the velocity?

The fraction

$$\frac{E_{i+1} - 2E_i + E_{i-1}}{h^2}$$

will also generate numbers from a normal distribution with mean zero, but this time with standard deviation  $2h^{-2}\sigma$ . Find out how much the noise is magnified in the computed acceleration signal.

The numbers in the `gauge.dat` file are given with 5 digits. This is no certain indication of the accuracy of the measurements, but as a test we may assume  $\sigma$  is of the order  $10^{-4}$ . Check if the visual results for the velocity and acceleration are consistent with the standard deviation of the noise in these signals as modeled above.  $\diamond$

**Exercise 8.47.** *Reduce the noise in Exer. 8.44.*

If we have a noisy signal  $\eta_i$ , where  $i = 0, \dots, n$  counts time levels, the noise can be reduced by computing a new signal where the value at a point is a weighted average of the values at that point and the neighboring points at each side. More precisely, given the signal  $\eta_i$ ,  $i = 0, \dots, n$ , we compute a filtered (averaged) signal with values  $\eta_i^{(1)}$  by the formula

$$\eta_i^{(1)} = \frac{1}{4}(\eta_{i+1} + 2\eta_i + \eta_{i-1}), \quad i = 1, \dots, n-1, \quad \eta_0^{(1)} = \eta_0, \quad \eta_n^{(1)} = \eta_n. \quad (8.19)$$

Make a function `filter` that takes the  $\eta_i$  values in an array `eta` as input and returns the filtered  $\eta_i^{(1)}$  values in an array. Let  $\eta_i^{(k)}$  be the signal arising by applying the `filtered` function  $k$  times to the same signal. Make a plot with curves  $\eta_i$  and the filtered  $\eta_i^{(k)}$  values for  $k = 1, 10, 100$ . Make similar plots for the velocity and acceleration where these are made from both the original  $\eta$  data and the filtered data. Discuss the results. Name of program file: `labstunami2.py`.  $\diamond$

**Exercise 8.48.** *Find the expected waiting time in traffic lights.*

A driver must pass 10 traffic lights on a certain route. Each light has a period red–yellow–green–yellow of two minutes, of which the green and yellow lights last for 70 seconds. Suppose the driver arrives at a traffic light at some uniformly distributed random point of time during

the period of two minutes. Compute the corresponding waiting time. Repeat this for 10 traffic lights. Run a large number of routes (i.e., repetitions of passing 10 traffic lights) and let the program write out the average waiting time. Does the computed time coincide with what you would expect? Name of program file: `waiting_time.py`. ◇