Random Numbers and Simple Games

8

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 basic programming concepts such as loops, lists, arrays, vectorization, curve plotting, and command-line arguments in order to study the present chapter. This means that Chaps. 1–5 of the present book should be digested. A few examples and exercises will require familiarity with the class concept from Chap. 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`[1].

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 $[0, 1)$ (recall that in the half open interval $[0, 1)$ the lower limit is included, but the upper limit is not). We can try it out:

[1] http://tinyurl.com/pwyasaa/random

H.P. Langtangen, *A Primer on Scientific Programming with Python*,
Texts in Computational Science and Engineering 6, DOI 10.1007/978-3-662-49887-3_8

```
>>> 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 8.20 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.

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 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 always recommend setting 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]

import scitools.std as st
st.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.

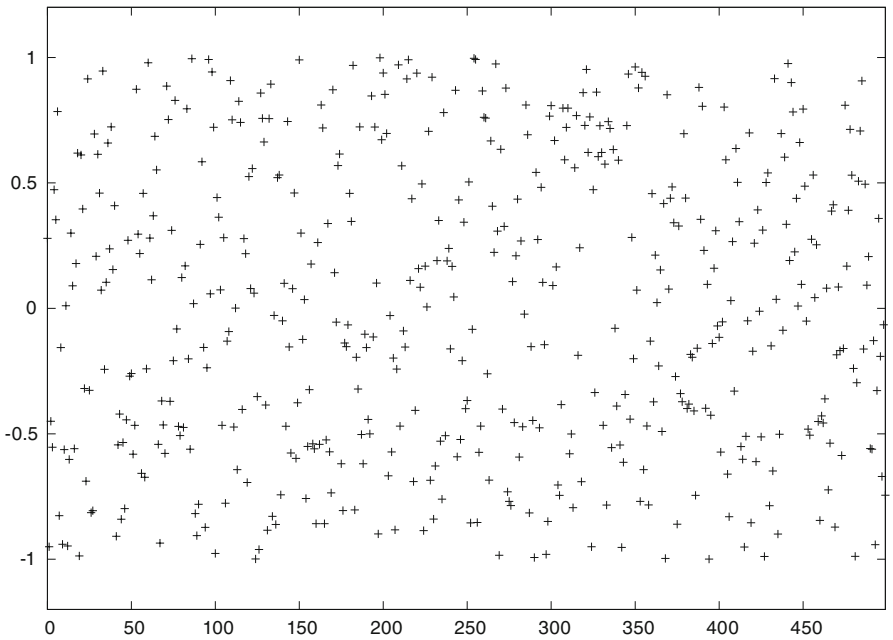

Fig. 8.1 The values of 500 random numbers drawn from the uniform distribution on $[-1, 1)$

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 $N \to \infty$. 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, $[a + ih, a + (i + 1)h]$, $i = 0, \ldots, n - 1$. 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 $N \to \infty$ and $n \to \infty$. 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 $N \to \infty$. 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 $N, n \to \infty$. 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 Fig. 8.2), called a *normalized histogram*. We can also define a piecewise constant function $p(x)$ from $H(i)$: $p(x) = H(i)$ for $x \in [a + ih, a + (i + 1)h)$, $i = 0, \ldots, n - 1$. As $n, N \to \infty$, $p(x)$ 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 $N \to \infty$, $p(x) \to 1$, which is illustrated by the plot.

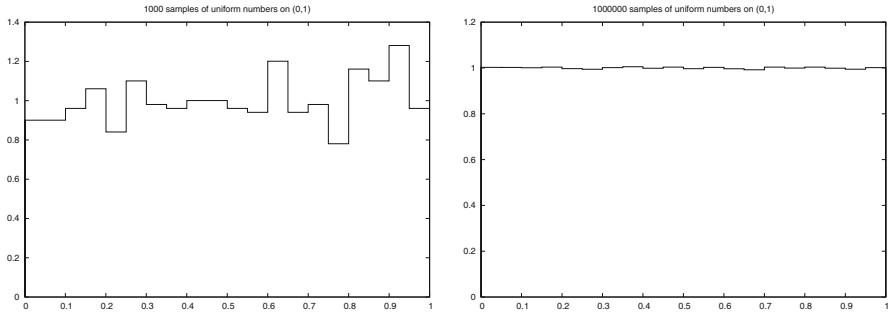

Fig. 8.2 The histogram of uniformly distributed random numbers in 20 bins

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:

```
import numpy as np
r = np.random.random()             # one number between 0 and 1
r = np.random.random(size=10000)   # array with 10000 numbers
r = np.random.uniform(-1, 10)      # one number between -1 and 10
r = np.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 in Sect. 8.5.3 and H.8.1.

Warning It is easy to do an `import random` followed by a `from numpy import *` or maybe `from scitools.std import *` without realizing that the latter two import statements import a name `random` from `numpy` 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, say

```
import random as random_number
```

Another solution is to do `import numpy as np` and work explicitly with `np.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 $x_0, x_1, \ldots, x_{n-1}$:

$$x_m = \frac{1}{n}\sum_{j=0}^{n-1} x_j \,. \tag{8.1}$$

The amount of spreading of the x_i values around the mean x_m can be measured by the *variance*,

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

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. A variant of (8.2) reads

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

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

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

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

$$x_m = s_m/q, \quad x_v = s_v/q - s_m^2/q^2 \,. \tag{8.5}$$

The *standard deviation*

$$x_s = \sqrt{x_v} \tag{8.6}$$

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 $x_0, \ldots, x_{n-1}$, 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 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
from math import sqrt
sm = 0; sv = 0
for q in range(1, N+1):
    x = random.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, see Sect. 3.4.2, for checking if a number can be divided by another without any remainder. 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 $N \to \infty$, equals $\sqrt{1/3} \approx 0.57735$.

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

```
import sys
N = int(sys.argv[1])
import numpy as np
x = np.random.uniform(-1, 1, size=N)
xm = np.mean(x)
xv = np.var(x)
xs = np.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. For example, the statistical distribution of the height or the blood pressure among adults of one gender are well described by a normal distribution. 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
r = random.normalvariate(m, s)
```

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

```
import numpy as np
r = np.random.normal(m, s, size=N)
r = np.random.randn(N)  # mean=0, std.dev.=1
```

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

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

import numpy as np
np.random.seed(12)
samples = np.random.normal(m, s, N)
print np.mean(samples), np.std(samples)

import scitools.std as st
x, y = st.compute_histogram(samples, 20, piecewise_constant=True)
st.plot(x, y, savefig='tmp.pdf',
        title ='%d samples of Gaussian/normal numbers on (0,1)' % N)
```

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 the famous, bell-shaped, normal distribution probability density function[2].

[2] http://en.wikipedia.org/wiki/Normal_distribution

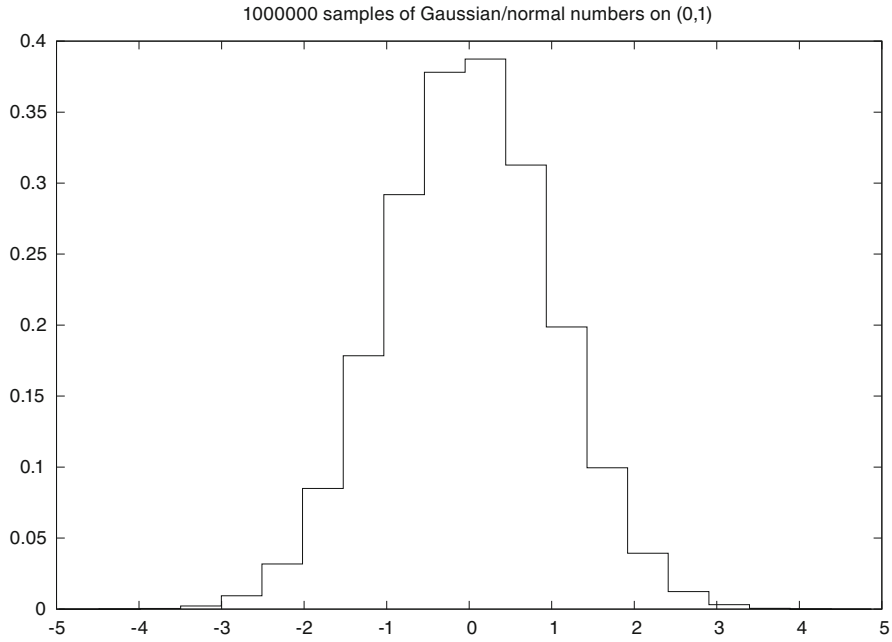

Fig. 8.3 Normalized histogram of 1 million random numbers drawn from the normal distribution

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 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
r = random.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`, . . ., `b-1`, `b`.

```
import random
r = random.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`, . . ., `b-1`, `b`, we must write

```
import numpy as np
r = np.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:

```
r = np.random.random_integers(a, b, N)
```

8.2.2 Example: Throwing a Die

Scalar version We can make a function that lets the computer throw a die `N` times and returns the fraction of the throws the die shows six eyes:

```
def six_eyes(N):
    M = 0                  # no of times we get 6 eyes
    for i in xrange(N):
        outcome = random.randint(1, 6)
        if outcome == 6:
            M += 1
    return float(M)/N
```

We use `xrange` instead of `range` because the former avoids storing `N` numbers in memory, which can be an important feature when `N` is large.

Vectorized version Too speed up the experiments, we can vectorize the drawing of the random numbers and the counting of the number successful experiments:

```
import numpy as np

def six_eyes_vec(N):
    eyes = np.random.randint(1, 7, N)
    success = eyes == 6     # True/False array
    M = np.sum(success)     # treats True as 1, False as 0
    return float(M)/N
```

The `eyes == 6` construction results in an array with `True` or `False` values, and `np.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 `np.sum` and not the standard `sum` function that is available in standard Python. With `np.sum` function, the vectorized version runs about 60 times faster than the scalar version. With Python's standard `sum` function, the vectorized versions is in fact twice as slow as the scalar version (!). We can illustrate the gain and loss in efficiency by the follow IPython session:

```
In [1]: from roll_die import six_eyes, six_eyes_vec

In [2]: %timeit six_eyes(100000)
1 loops, best of 3: 250 ms per loop

In [3]: %timeit six_eyes_vec(100000)
100 loops, best of 3: 4.11 ms per loop

In [4]: 250/4.11           # performance fraction
Out[4]: 60.8272506082725

In [5]: from roll_die import np

In [6]: np.sum = sum  # fool numpy to use built-in Python sum

In [7]: %timeit six_eyes_vec(100000)
1 loops, best of 3: 543 ms per loop
```

(Note how we can bind Python's built-in `sum` function to the `np.sum` name such that `np.sum` in `six_eyes_vec` applies Python's `sum` function instead of the original `np.sum`.)

Vectorized version with batches The disadvantage with the vectorized version is that all the random numbers must be stored in the computer's memory. A large `N` may cause the program to run out of memory and raise a `MemoryError`. Instead of drawing all random numbers at once, we can draw them in batches of size `arraysize`. There will be `N//arraysize` such batches, plus a rest. Note the double slash in `N//arraysize`: here we indeed want integer division, which is explicitly instructed in Python by the double forward slash. The rest is obtained by the mod operator: `rest = N % arraysize`. The size of the batches can be stored in a list:

```
rest = N % arraysize
batch_sizes = [arraysize]*(N//arraysize) + [rest]
```

We can now make one batch of random numbers at a time and count how many times we get six:

```
def six_eyes_vec2(N, arraysize=1000000):
    # Split all experiments into batches of size arraysize,
    # plus a final batch of size rest
    # (note: N//arraysize is integer division)
    rest = N % arraysize
    batch_sizes = [arraysize]*(N//arraysize) + [rest]

    M = 0
    for batch_size in batch_sizes:
        eyes = np.random.randint(1, 7, batch_size)
        success = eyes == 6      # True/False array
        M += np.sum(success)     # treats True as 1, False as 0
    return float(M)/N
```

Because we fix the seed, the computed `f` will always be the same in this function.

Verification of the scalar version Verifying computations with random numbers requires the seed to be fixed. When we believe the scalar version in function `six_eyes` is correct, mainly by observing that the return value approaches 1/6 as the number of experiments, `N`, grows, we can call the function with a small `N` and record the return value for use in a test function:

```
def test_scalar():
    random.seed(3)
    f = six_eyes(100)
    f_exact = 0.26
    assert abs(f_exact - f) < 1E-15
```

Verification of all versions Since we have three alternative functions for computing the same quantity, a verification can be based on comparing the output of all three functions. This is somewhat problematic since the scalar and vectorized versions apply different random number generators. Fixing the seed of Python's `random` module and `numpy.random` does not help as these two tools will generate different sequences of random integers. Nevertheless, we can fool the scalar version in the `six_eyes` function to use `np.random.randint` instead of `random.randit`: this is just a matter of setting `random = np.random` (with a declaration `global random`) before calling `six_eyes`. The problem is that the call `np.random.randint(1, 6)` in `six_eyes` will then generate the numbers up to *but not including* 6, so `M` will always be zero. A little trick, can solve the problem: we redefine `random.randit` to be a function that calls `np.random.randint`:

```
random.randint = lambda a, b: np.random.randint(a, b+1, 1)[0]
```

The call `random.randint(1, 6)` in `six_eyes` now becomes `np.random.randint(1, 7, 1)[0]`, i.e., we generate an array of 1 random integer and extract the first element so the result is a scalar number as before.

A test function can call all three functions, with the same fixed seed, and compare the returned values:

```
def test_all():
    # Use np.random as random number generator for all three
    # functions and make sure all of them applies the same seed
    N = 100
    arraysize = 40
    random.randint = lambda a, b: np.random.randint(a, b+1, 1)[0]
    tol = 1E-15

    np.random.seed(3)
    f_scalar = six_eyes(N)
    np.random.seed(3)
    f_vec = six_eyes_vec(N)
    assert abs(f_scalar - f_vec) < tol

    np.random.seed(3)
    f_vec2 = six_eyes_vec2(N, arraysize=80)
    assert abs(f_vec - f_vec2) < tol
```

All the functions above are found in the file `roll_die.py`[3].

8.2.3 Drawing a Random Element from a List

Given a list `a`, the statement

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

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

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

There is also a function `shuffle` that performs a random permutation of the list elements:

```
random.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
>>> random.choice(awards)
'car'
```

[3] http://tinyurl.com/pwyasaa/random/roll_die

```
>>> awards[random.randint(0, len(awards)-1)]
'pen'
>>> random.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.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`, `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 data 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 data attribute, and methods for dealing one or several hands and for putting back a card:

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

class Hand(object):
    ...
    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 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 $N \to \infty$. (Note that in this limit, $M \to \infty$ 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*. (Note that this term is also applied for programs that solve equations arising in mathematical models in general, but it is even more common to use the term when random numbers are used to estimate probabilities.) The mathematical technique of letting the computer perform lots of experiments 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.

> *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.

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. Sections 8.3.2–8.3.5 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 Sect. 8.5).

It appears that Monte Carlo simulation programmed in pure Python is a computationally feasible approach, even on small laptops, in all the forthcoming examples. Significant speed-up can be achieved by vectorizing the code, which is explained in detail for many of the examples. However, large-scale Monte Carlo simulations and other heavy computations run slowly in pure Python, and the core of the computations should be moved to a compiled language such as C. In Appendix G, you can find a Monte Carlo application that is implemented in pure Python, in vectorized `numpy` Python, in the extended (and very closely related) Cython language, as well as in pure C code. Various ways of combining Python with C are also illustrated.

8.3.2 Example: Throwing Dice

You throw two dice, one black and one green. What is the probability that the number of eyes on the black die is larger than the number of eyes on the green die?

Straightforward solution We can simulate N throws of two dice in a program. For each throw we see if the event is successful, and if so, increase M by one:

```
import sys
N = int(sys.argv[1])              # no of experiments

import random
M = 0                             # no of successful events
```

```
for i in range(N):
    black = random.randint(1, 6)   # throw black
    green = random.randint(1, 6)   # throw brown
    if black > green:              # success?
        M += 1
p = float(M)/N
print 'probability:', p
```

This program is named black_gt_green.py.

Vectorization Although the black_gt_green.py program runs $N = 10^6$ in a few seconds, Monte Carlo simulation programs can quickly require quite some simulation time so speeding up the algorithm by vectorization is often desired. Let us vectorize the code shown above. The idea is to draw all the random numbers $(2N)$ at once. We make an array of random numbers between 1 and 6 with 2 rows and N columns. The first row can be taken as the number of eyes on the black die in all the experiments, while the second row are the corresponding eyes on the green die:

```
r = np.random.random_integers(1, 6, size=(2, N))
black = r[0,:]            # eyes for all throws with black
green = r[1,:]            # eyes for all throws with green
```

The condition black > green results in an array of length N of boolean values: True when the element in black is greater than the corresponding element in green, and False if not. The number of True elements in the boolean array black > green is then M. This number can be computed by summing up all the boolean values. In arithmetic operations, True is 1 and False i 0, so the sum equals M. Fast summation of arrays requires np.sum and not Python's standard sum function. The code goes like

```
success = black > green   # success[i] is true if black[i]>green[i]
M = np.sum(success)       # sum up all successes
p = float(M)/N
print 'probability:', p
```

The code, found in the file black_gt_green_vec.py, runs over 10 times faster than the corresponding scalar code in black_gt_green.py.

Exact solution In this simple example we can quite easily compute the exact solution. To this end, we set up all the outcomes of the experiment, i.e., all the possible combinations of eyes on two dice:

```
combinations = [(black, green)
                for black in range(1, 7)
                for green in range(1, 7)]
```

Then we count how many of the (black, green) pairs that have the property black > green:

```
success = [black > green for black, green in combinations]
M = sum(success)
```

It turns out that `M` is 15, giving a probability $15/36 \approx 0.41667$ since there are 36 combinations in total. Running the Monte Carlo simulations with $N = 10^6$ typically gives probabilities in $[0.416, 0.417]$.

A game Suppose a games is constructed such that you have to pay 1 euro to throw the two dice. You win 2 euros if there are more eyes on the black than on the green die. Should you play this game? We can easily simulate the game directly (file `black_gt_green_game.py`):

```
import sys
N = int(sys.argv[1])              # no of experiments

import random
start_capital = 10
money = start_capital
for i in range(N):
    money -= 1                    # pay for the game
    black = random.randint(1, 6)  # throw black
    green = random.randint(1, 6)  # throw brown
    if black > green:             # success?
        money += 2                # get award

net_profit_total = money - start_capital
net_profit_per_game = net_profit_total/float(N)
print 'Net profit per game in the long run:', net_profit_per_game
```

Experimenting with a few N shows that the net profit per game is always negative. That is, you should *not* play this game.

A vectorized version is beneficial of efficiency reasons (the corresponding file is `black_gt_green_game_vec.py`):

```
import sys
N = int(sys.argv[1])      # no of experiments

import numpy as np
r = np.random.random_integers(1, 6, size=(2, N))

money = 10 - N            # capital after N throws
black = r[0,:]            # eyes for all throws with black
green = r[1,:]            # eyes for all throws with green
success = black > green   # success[i] is true if black[i]>green[i]
M = np.sum(success)       # sum up all successes
money += 2*M              # add all awards for winning
print 'Net profit per game in the long run:', (money-10)/float(N)
```

Decide if a game is fair Suppose the cost of playing a game once is q and that the award for winning is r. The net income in a winning game is $r - q$. Winning

M out of N games means that the cost is Nq and the income is Mr, making a net profit $s = Mr - Nq$. Now $p = M/N$ is the probability of winning the game so $s = (pr - q)N$. A fair game means that we neither win nor lose in the long run: $s = 0$, which implies that $r = q/p$. That is, given the cost q and the probability p of winning, the award paid for winning the game must be $r = q/p$ in a fair game.

When somebody comes up with a game you can use Monte Carlo simulation to estimate p and then conclude that you should not play the game of $r < q/p$. The example above has $p = 15/36$ (exact) and $q = 1$, so $r = 2.4$ makes a fair game.

The reasoning above 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 the probabilities times the corresponding net income for each event: $-q(1 - p) + (r - q)p$ (recall that the net income in a winning game is $r - q$). A fair game has zero expected payment, which leads to $r = q/p$.

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
color = random.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.choice(hat)
    hat.remove(color)
    return color, hat
```

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

def draw_ball(hat):
    index = random.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

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

def draw_ball(hat):
    """Draw a ball using list index."""
    index = random.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.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 8.21 asks you to design and implement a class `Hat`.

8.3.4 Random Mutations of Genes

A simple mutation model A fundamental principle of biological evolution is that DNA undergoes mutation. Since DNA can be represented as a string consisting of the letters A, C, G, and T, as explained in Sect. 3.3, mutation of DNA is easily modeled by replacing the letter in a randomly chosen position of the DNA by a randomly chosen letter from the alphabet A, C, G, and T. A function for replacing the letter in a randomly selected position (index) by a random letter among A, C, G, and T is most straightforwardly implemented by converting the DNA string to a list of letters, since changing a character in a Python string is impossible without constructing a new string. However, an element in a list can be changed in-place:

```
import random

def mutate_v1(dna):
    dna_list = list(dna)
    mutation_site = random.randint(0, len(dna_list) - 1)
    dna_list[mutation_site] = random.choice(list('ATCG'))
    return ''.join(dna_list)
```

Using `get_base_frequencies_v2` and `format_frequencies` from Sect. 6.5.3, we can easily mutate a gene a number of times and see how the frequencies of the bases A, C, G, and T change:

```
dna = 'ACGGAGATTTCGGTATGCAT'
print 'Starting DNA:', dna
print format_frequencies(get_base_frequencies_v2(dna))

nmutations = 10000
for i in range(nmutations):
    dna = mutate_v1(dna)

print 'DNA after %d mutations:' % nmutations, dna
print format_frequencies(get_base_frequencies_v2(dna))
```

Here is the output from a run:

```
Starting DNA: ACGGAGATTTCGGTATGCAT
A: 0.25, C: 0.15, T: 0.30, G: 0.30
DNA after 10000 mutations: AACCAATCCGACGAGGAGTG
A: 0.35, C: 0.25, T: 0.10, G: 0.30
```

Vectorized version The efficiency of the `mutate_v1` function with its surrounding loop can be significantly increased up by performing all the mutations at once using `numpy` arrays. This speed-up is of interest for long `dna` strings and many mutations. The idea is to draw all the mutation sites at once, and also all the new bases at these sites at once. The `np.random` module provides functions for drawing several random numbers at a time, but only integers and real numbers can be drawn, not characters from the alphabet A, C, G, and T. We therefore have to simulate these four characters by the numbers (say) 0, 1, 2, and 3. Afterwards we can translate the integers to letters by some clever vectorized indexing.

Drawing `N` mutation sites is a matter of drawing `N` random integers among the legal indices:

```
import numpy as np
mutation_sites = np.random.random_integers(0, len(dna)-1, size=N)
```

Drawing `N` bases, represented as the integers 0–3, is similarly done by

```
new_bases_i = np.random.random_integers(0, 3, N)
```

Converting say the integers 1 to the base symbol C is done by picking out the indices (in a boolean array) where `new_bases_i` equals 1, and inserting the character `'C'` in a companion array of characters:

```
new_bases_c = np.zeros(N, dtype='c')
indices = new_bases_i == 1
new_bases_c[indices] = 'C'
```

We must do this integer-to-letter conversion for all four integers/letters. Thereafter, `new_bases_c` must be inserted in `dna` for all the indices corresponding to the randomly drawn mutation sites,

```
dna[mutation_sites] = new_bases_c
```

The final step is to convert the `numpy` array of characters `dna` back to a standard string by first converting `dna` to a list and then joining the list elements: `''.join(dna.tolist())`.

The complete vectorized function can now be expressed as follows:

```
import numpy as np
# Use integers in random numpy arrays and map these
# to characters according to
i2c = {0: 'A', 1: 'C', 2: 'G', 3: 'T'}

def mutate_v2(dna, N):
    dna = np.array(dna, dtype='c')  # array of characters
    mutation_sites = np.random.random_integers(
        0, len(dna) - 1, size=N)
    # Must draw bases as integers
    new_bases_i = np.random.random_integers(0, 3, size=N)
    # Translate integers to characters
    new_bases_c = np.zeros(N, dtype='c')
    for i in i2c:
        new_bases_c[new_bases_i == i] = i2c[i]
    dna[mutation_sites] = new_bases_c
    return ''.join(dna.tolist())
```

It is of interest to time `mutate_v2` versus `mutate_v1`. For this purpose we need a long test string. A straightforward generation of random letters is

```
def generate_string_v1(N, alphabet='ACGT'):
    return ''.join([random.choice(alphabet) for i in xrange(N)])
```

A vectorized version of this function can also be made, using the ideas explored above for the `mutate_v2` function:

```
def generate_string_v2(N, alphabet='ACGT'):
    # Draw random integers 0,1,2,3 to represent bases
    dna_i = np.random.random_integers(0, 3, N)
```

```
    # Translate integers to characters
    dna = np.zeros(N, dtype='c')
    for i in i2c:
        dna[dna_i == i] = i2c[i]
    return ''.join(dna.tolist())
```

The `time_mutate` function in the file `mutate.py` performs timing of the generation of test strings and the mutations. To generate a DNA string of length 100,000 the vectorized function is about 8 times faster. When performing 10,000 mutations on this string, the vectorized version is almost 3000 times faster! These numbers stay approximately the same also for larger strings and more mutations. Hence, this case study on vectorization is a striking example on the fact that a straightforward and convenient function like `mutate_v1` might occasionally be very slow for large-scale computations.

A Markov chain mutation model The observed rate at which mutations occur at a given position in the genome is not independent of the type of nucleotide (base) at that position, as was assumed in the previous simple mutation model. We should therefore take into account that the rate of transition depends on the base.

There are a number of reasons why the observed mutation rates vary between different nucleotides. One reason is that there are different mechanisms generating transitions from one base to another. Another reason is that there are extensive repair process in living cells, and the efficiency of this repair mechanism varies for different nucleotides.

Mutation of nucleotides may be modeled using distinct probabilities for the transitions from each nucleotide to every other nucleotide. For example, the probability of replacing A by C may be prescribed as (say) 0.2. In total we need 4×4 probabilities since each nucleotide can transform into itself (no change) or three others. The sum of all four transition probabilities for a given nucleotide must sum up to one. Such statistical evolution, based on probabilities for transitioning from one state to another, is known as a Markov process or Markov chain.

First we need to set up the probability matrix, i.e., the 4×4 table of probabilities where each row corresponds to the transition of A, C, G, or T into A, C, G, or T. Say the probability transition from A to A is 0.2, from A to C is 0.1, from A to G is 0.3, and from A to T is 0.4.

Rather than just prescribing some arbitrary transition probabilities for test purposes, we can use random numbers for these probabilities. To this end, we generate three random numbers to divide the interval $[0, 1]$ into four intervals corresponding to the four possible transitions. The lengths of the intervals give the transition probabilities, and their sum is ensured to be 1. The interval limits, 0, 1, and three random numbers must be sorted in ascending order to form the intervals. We use the function `random.random()` to generate random numbers in $[0, 1)$:

```
slice_points = sorted(
    [0] + [random.random() for i in range(3)] + [1])
transition_probabilities = [slice_points[i+1] - slice_points[i]
                            for i in range(4)]
```

The transition probabilities are handy to have available as a dictionary:

```
markov_chain['A'] = {'A': ..., 'C': ..., 'G': ..., 'T': ...}
```

which can be computed by

```
markov_chain['A'] = {base: p for base, p in
                     zip('ACGT', transition_probabilities)}
```

To select a transition, we need to draw a random letter (A, C, G, or T) according to the probabilities `markov_chain[b]` where `b` is the base at the current position. Actually, this is a very common operation, namely drawing a random value from a *discrete probability distribution* (`markov_chain[b]`). The natural approach is therefore write a general function for drawing from any discrete probability distribution given as a dictionary:

```
def draw(discrete_probdist):
    """
    Draw random value from discrete probability distribution
    represented as a dict: P(x=value) = discrete_probdist[value].
    """
    # Method:
    # http://en.wikipedia.org/wiki/Pseudo-random_number_sampling
    limit = 0
    r = random.random()
    for value in discrete_probdist:
        limit += discrete_probdist[value]
        if r < limit:
            return value
```

Basically, the algorithm divides [0, 1] into intervals of lengths equal to the probabilities of the various outcomes and checks which interval is hit by a random variable in [0, 1]. The corresponding value is the random choice.

A complete function creating all the transition probabilities and storing them in a dictionary of dictionaries takes the form

```
def create_markov_chain():
    markov_chain = {}
    for from_base in 'ATGC':
        # Generate random transition probabilities by dividing
        # [0,1] into four intervals of random length
       slice_points = sorted(
           [0] + [random.random()for i in range(3)] + [1])
       transition_probabilities = \
           [slice_points[i+1] - slice_points[i] for i in range(4)]
       markov_chain[from_base] = {base: p for base, p
                          in zip('ATGC', transition_probabilities)}
    return markov_chain

mc = create_markov_chain()
print mc
print mc['A']['T'] # probability of transition from A to T
```

It is natural to develop a function for checking that the generated probabilities are consistent. The transition from a particular base into one of the four bases happens with probability 1, which means that the probabilities in a row must sum up to 1:

```
def check_transition_probabilities(markov_chain):
    for from_base in 'ATGC':
        s = sum(markov_chain[from_base][to_base]
                for to_base in 'ATGC')
        if abs(s - 1) > 1E-15:
            raise ValueError('Wrong sum: %s for "%s"' % \
                             (s, from_base))
```

Another test is to check that `draw` actually draws random values in accordance with the underlying probabilities. To this end, we draw a large number of values, `N`, count the frequencies of the various values, divide by `N` and compare the empirical normalized frequencies with the probabilities:

```
def check_draw_approx(discrete_probdist, N=1000000):
    """
    See if draw results in frequencies approx equal to
    the probability distribution.
    """
    frequencies = {value: 0 for value in discrete_probdist}
    for i in range(N):
        value = draw(discrete_probdist)
        frequencies[value] += 1
    for value in frequencies:
        frequencies[value] /= float(N)
    print ', '.join(['%s: %.4f (exact %.4f)' % \
                     (v, frequencies[v], discrete_probdist[v])
                     for v in frequencies])
```

This test is only approximate, but does bring evidence to the correctness of the implementation of the `draw` function.

A vectorized version of `draw` can also be made. We refer to the source code file `mutate.py` for details (the function is relatively complicated).

Now we have all the tools needed to run the Markov chain of transitions for a randomly selected position in a DNA sequence:

```
def mutate_via_markov_chain(dna, markov_chain):
    dna_list = list(dna)
    mutation_site = random.randint(0, len(dna_list) - 1)
    from_base = dna[mutation_site]
    to_base = draw(markov_chain[from_base])
    dna_list[mutation_site] = to_base
    return ''.join(dna_list)
```

Exercise 8.47 suggests some efficiency enhancements of simulating mutations via these functions.

Here is a simulation of mutations using the method based on Markov chains:

```
dna = 'TTACGGAGATTTCGGTATGCAT'
print 'Starting DNA:', dna
print format_frequencies(get_base_frequencies_v2(dna))

mc = create_markov_chain()
import pprint
print 'Transition probabilities:\n', pprint.pformat(mc)
nmutations = 10000
for i in range(nmutations):
    dna = mutate_via_markov_chain(dna, mc)

print 'DNA after %d mutations (Markov chain):' % nmutations, dna
print format_frequencies(get_base_frequencies_v2(dna))
```

The output will differ each time the program is run unless `random.seed(i)` is called in the beginning of the program for some integer `i`. This call makes the sequence of random numbers the same every time the program is run and is very useful for debugging. An example on the output may look like

```
Starting DNA: TTACGGAGATTTCGGTATGCAT
A: 0.23, C: 0.14, T: 0.36, G: 0.27
Transition probabilities:
{'A': {'A': 0.4288890546751146,
       'C': 0.4219086988655296,
       'G': 0.00668870644455688,
       'T': 0.14251354001479888},
 'C': {'A': 0.24999667668640035,
       'C': 0.04718309085408834,
       'G': 0.6250440975238185,
       'T': 0.0777761349356928},
 'G': {'A': 0.16022955651881965,
       'C': 0.34652746609882423,
       'G': 0.1328031742612512,
       'T': 0.3604398031211049},
 'T': {'A': 0.20609823213950174,
       'C': 0.17641112746655452,
       'G': 0.010267621176125452,
       'T': 0.6072230192178183}}
DNA after 10000 mutations (Markov chain): GGTTTAAGTCAGCTATGATTCT
A: 0.23, C: 0.14, T: 0.41, G: 0.23
```

The various functions performing mutations are located in the file `mutate.py`.

8.3.5 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 over-representation 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`:

```
import numpy as np
r = np.random.random(n)
parents = np.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 = np.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 = np.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 = np.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 a fraction 0.01 of the population does not follow the one son policy and get 4 children on average, the population grows with a factor 1.5 over 10 generations (`male_portion` and `fertility` kept at the ideal values 0.5 and 1, respectively).

Normally, simple models like the difference equations (A.9) and (A.12), from Sects. A.1.4 and A.1.5, or the differential equations (C.11) or (C.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 8.26 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
number = random.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 guess_number.py. Try it out! Can you come up with a strategy for reducing the number of attempts? See Exercise 8.27 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 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

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

def computer_guess(ndice):
    return random.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):
    player_capital = computer_capital = nrounds  # start capital

    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)

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

        print 'Machine guessed %d, got %d' % (guess, throw)

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

        if player_capital == 0 or computer_capital == 0:
            break

    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 8.12 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(object):
    def __init__(self, n=1):
        self.n = n   # no of dice

    def throw(self):
        return [random.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(object):
    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):
    # All guesses have the same probability
    return random.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 has %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, which is actually non-random (deterministic) problem. Computing integrals with the aid of random numbers is known as Monte Carlo integration and is one of the most powerful and widely used mathematical technique throughout science and engineering.

Our main focus here will be integrals of the type $\int_a^b f(x)dx$ for which the Monte Carlo integration is not a competitive technique compared to simple methods such as the Trapezoidal method, the Midpoint method, or Simpson's method. However, for integration of functions of *many variables*, the Monte Carlo approach is the best method we have. Such integrals arise all the time in quantum physics, financial engineering, and when estimating the uncertainty of mathematical computations. What you learn about Monte Carlo integration of functions of one variable ($\int_a^b f(x)dx$) is directly transferable to the important application cases where there are many variables involved.

8.5.1 Derivation of Monte Carlo Integration

There are two ways to introduce Monte Carlo integration, one based on calculus and one based on probability theory. The goal is to compute a numerical approximation to

$$\int_a^b f(x)dx\,.$$

The calculus approach via the mean-value theorem The mean-value theorem from calculus states that

$$\int_a^b f(x)dx = (b-a)\bar{f}, \tag{8.7}$$

where $\bar{f}$ is the mean value of f, defined as

$$\bar{f} = \frac{1}{b-a}\int_a^b f(x)dx\,.$$

One way of using (8.7) to define a numerical method for integration is to approximate $\bar{f}$ by taking the average of f at n points $x_0, \ldots, x_{n-1}$:

$$\bar{f} \approx \frac{1}{n}\sum_{i=0}^{n-1} f(x_i)\,. \tag{8.8}$$

(We let the numbering of the points go from 0 to $n-1$ because these numbers will later be indices in Python arrays, which have to start at 0.)

There is freedom in how to choose the points $x_0, \ldots, x_{n-1}$. We could, for example, make them uniformly spaced in the interval $[a, b]$. The particular choice

$$x_0 = a + ih + \frac{1}{2}h, \quad i = 0, \ldots, n-1, \quad h = \frac{b-a}{n-1},$$

correspond to the famous Midpoint rule for numerical integration. Intuitively, we anticipate that the more points we use, the better is the approximation $\frac{1}{n}\sum_i f(x_i)$ to the exact mean value $\bar{f}$. For the Midpoint rule one can show mathematically, or numerically estimate through examples, that the error in the approximation depends on n as n^{-2}. That is, doubling the number of points reduces the error by a factor 1/4.

A slightly different set of uniformly distributed points is

$$x_i = a + ih, \quad i = 0, \ldots, n-1, \ldots, \quad h = \frac{b-1}{n-2}\,.$$

These points might look more intuitive to many since they start at a and end at b ($x_0 = a$, $x_{n-1} = b$), but the error in the integration rule now goes as n^{-1}: doubling the number of points just halves the error. That is, computing more function values to get a better integration estimate is less effective with this set points than with the slight displaced points used in the Midpoint rule.

One could also throw in the idea of using a set of *random* points, uniformly distributed in $[a, b]$. This is the Monte Carlo integration technique. The error now goes like $n^{-1/2}$, which means that many more points and corresponding function evaluations are needed to reduce the error compared to using the Midpoint method. The surprising fact, however, is that using random points for many variables (in high-dimensional vector spaces) yields a very effective integration technique, dramatically more effective than extending the ideas of the Midpoint rule to many variables.

The probability theory approach People who are into probability theory usually like to interpret integrals as mathematical expectations of a random variable. (If you are not one of those, you can safely jump to Sect. 8.5.2 where we just program the simple sum in the Monte Carlo integration method.) More precisely, the integral $\int_a^b f(x)dx$ can be expressed as a mathematical expectation of $f(x)$ if x is a uniformly distributed random variable on $[a, b]$. This expectation can be estimated by the average of random samples, which results in the Monte Carlo integration method. To see this, we start with the formula for the probability density function for a uniformly distributed random variable X on $[a, b]$:

$$p(x) = \begin{cases} (b-a)^{-1}, & x \in [a,b] \\ 0, & \text{otherwise} \end{cases}$$

Now we can write the standard formula for the mathematical expectation $\mathrm{E}(f(X))$:

$$\mathrm{E}(f(X)) = \int_{-\infty}^{\infty} f(x)p(x)dx = \int_a^b f(x)\frac{1}{b-a}dx = (b-a)\int_a^b f(x)dx\,.$$

The last integral is exactly what we want to compute. An expectation is usually estimated from a lot of samples, in this case uniformly distributed random numbers $x_0, \ldots, x_{n-1}$ in $[a, b]$, and computing the sample mean:

$$\mathrm{E}(f(X)) \approx \frac{1}{n}\sum_{i=0}^{n-1} f(x_i)\,.$$

The integral can therefore be estimated by

$$\int_a^b f(x)dx \approx (b-a)\frac{1}{n}\sum_{i=0}^{n-1} f(x_i),$$

which is nothing but the Monte Carlo integration method.

8.5.2 Implementation of Standard Monte Carlo Integration

To summarize, Monte Carlo integration consists in generating n uniformly distributed random numbers x_i in $[a, b]$ and then compute

$$(b-a)\frac{1}{n}\sum_{i=0}^{n-1} f(x_i)\,. \tag{8.9}$$

We can implement (8.9) in a small function:

```
import random

def MCint(f, a, b, n):
    s = 0
    for i in xrange(n):
        x = random.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 very useful:

```
import numpy as np

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

The functions above are available in the module file `MCint.py`. We can test the gain in efficiency with `%timeit` in an IPython session:

```
In [1]: from MCint import MCint, MCint_vec

In [2]: from math import sin, pi

In [3]: %timeit MCint(sin, 0, pi, 1000000)
1 loops, best of 3: 1.19 s per loop

In [4]: from numpy import sin

In [5]: %timeit MCint_vec(sin, 0, pi, 1000000)
1 loops, best of 3: 173 ms per loop

In [6]: 1.19/0.173          # relative performance
Out[6]: 6.878612716763006
```

Note that we use `sin` from `math` in the scalar function `MCint` because this function is significantly faster than `sin` from `numpy`:

```
In [7]: from math import sin

In [8]: %timeit sin(1.2)
10000000 loops, best of 3: 179 ns per loop

In [9]: from numpy import sin

In [10]: %timeit sin(1.2)
100000 loops, best of 3: 3.22 microsec per loop

In [11]: 3.22E-6/179E-9      # relative performance
Out[11]: 17.988826815642458
```

(A similar test reveals that `math.sin` is 1.3 times slower calling `sin` without a prefix. The differences are much smaller between `numpy.sin` and the same function without the prefix.)

The increase in efficiency by using `MCint_vec` instead of `MCint` is in the test above a factor of 6–7, which is not dramatic. Moreover, the vectorized version needs to store n random numbers and n function values in memory. A better vectorized implementation for large n is to split the `x` and `f(x)` arrays into chunks of a given size `arraysize` such that we can control the memory usage. Mathematically, it means to split the sum $\frac{1}{n}\sum_i f(x_i)$ into a sum of smaller sums. An appropriate implementation reads

```
def MCint_vec2(f, a, b, n, arraysize=1000000):
    s = 0
    # Split sum into batches of size arraysize
    # + a sum of size rest (note: n//arraysize is integer division)
    rest = n % arraysize
    batch_sizes = [arraysize]*(n//arraysize) + [rest]
    for batch_size in batch_sizes:
        x = np.random.uniform(a, b, batch_size)
        s += np.sum(f(x))
    I = (float(b-a)/n)*s
    return I
```

With 100 million points, `MCint_vec2` is about 10 faster than `MCint`. (Note that the latter function must use `xrange` and not `range` for so large n, otherwise the array returned by `range` may become to large to be stored in memory in a small computer. The `xrange` function generates one `i` at a time without the need to store all the `i` values.)

Example 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 with 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 = np.zeros(n)
    for k in range(1, n+1):
        x = random.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`, such that `k` reflects the actual number of points used in the method. 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 be computed by the mod function: `k % N` gives the remainder when `k` is divided by `N`. In our case we can store when this reminder is zero,

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

This recipe of doing something every N-th pass in long loops has lots of applications in scientific computing! The complete function now 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.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

def demo():
```

Our sample application goes like

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

k, I = MCint3(f1, 1, 2, 1000000, N=10000)
error = 6.5 - np.array(I)
```

Figure 8.4 shows a plot of `error` versus the number of function evaluation `k`.

Remark We claimed that the Monte Carlo method is slow for integration functions of one variable. There are, however, many techniques that apply smarter ways of drawing random numbers, so called variance reduction techniques, and thereby increase the computational efficiency.

8.5.3 Area Computing 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,

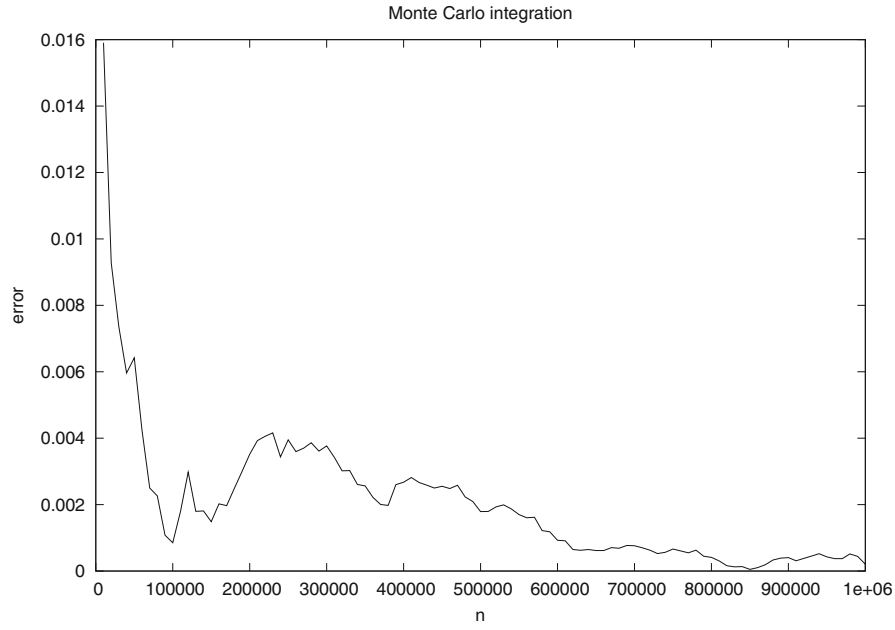

Fig. 8.4 The convergence of Monte Carlo integration applied to $\int_1^2 (2+3x)dx$

$(x_H - x_L)(y_H - y_L)$. 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,

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

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 Fig. 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.uniform(a, b)
        y = random.uniform(0, m)
        if y <= f(x):
            below += 1
    area = below/float(n)*m*(b-a)
    return area
```

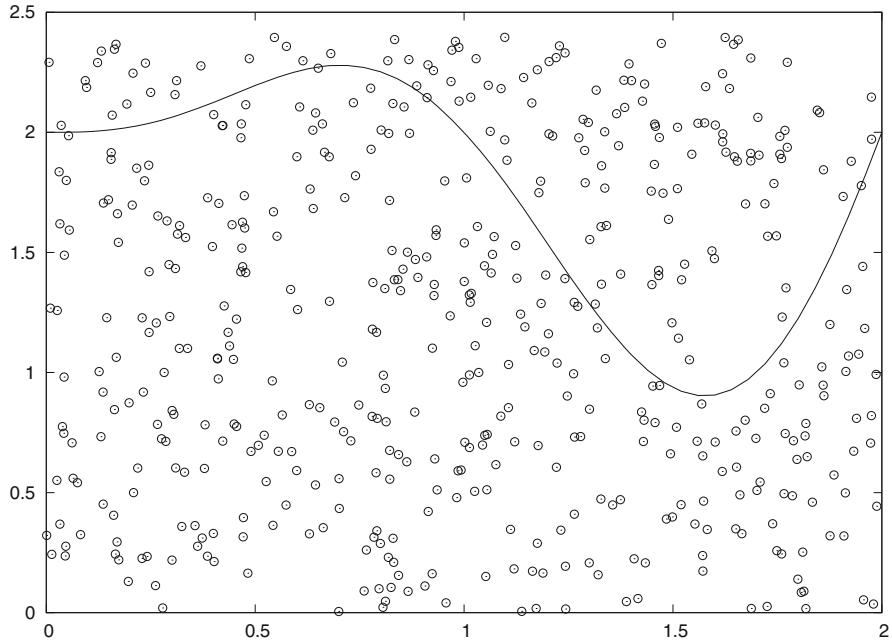

Fig. 8.5 The "dart" method for computing integrals. When M out of N random points in the rectangle $[0, 2] \times [0, 2.4]$ lie under the curve, the area under the curve is estimated as the M/N fraction of the area of the rectangle, i.e., $(M/N)2 \cdot 2.4$

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

A vectorized implementation reads

```
import numpy as np

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

The only non-trivial line here is the expression `y[y < f(x)]`, which applies boolean indexing (Sect. 5.5.2) to extract the `y` values that are below the `f(x)` curve. The sum of the boolean values (interpreted as 0 and 1) in `y < f(x)` gives the number of points below the curve.

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 Sect. H.8.1):

```
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 $n = 10^6$ the author achieved a factor of about 8 in favor of the vectorized version.

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

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 8.32. 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 8.33 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
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.randint(1,2)  # flip coin
        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:

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

These two statements require `from scitools.std import plot` and `import time`.

It is very important that the extent of the axis are kept fixed in animations, otherwise one gets a wrong visual impression. 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. To be specific, the probability is 2^{-n_s}, which becomes about 10^{-9} for 30 steps. 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 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 Sect. 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 (see Appendix A for an introduction to difference equations). Let x_n be the position of the particle at 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 $\mathrm{P}(A)$. We can therefore write $\mathrm{P}(s = 1) = 1/2$ and $\mathrm{P}(s = -1) = 1/2$. The difference equation can now be expressed mathematically as

$$x_n = x_{n-1} + s, \quad x_0 = 0, \quad \mathrm{P}(s = 1) = \mathrm{P}(s = -1) = 1/2\,. \tag{8.10}$$

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 (8.10), 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 Sect. 8.1.5):

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

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

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:

```
    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 ymax on the maximum value of the histogram (max(freq)), plus some space (chosen as 10 percent of max(freq)). However, we do not change the ymax value unless it is more than 0.1 different from the previous ymax value (otherwise the axis "jumps" too often). The minimum value, ymin, is set to ymin=-0.1*ymax every time we change the ymax 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 Fig. 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 Δx, 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 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's random module. 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 = numpy.random.randint(1, 3, size=np*ns)
# or
moves = numpy.random.random_integers(1, 2, size=np*ns)
```

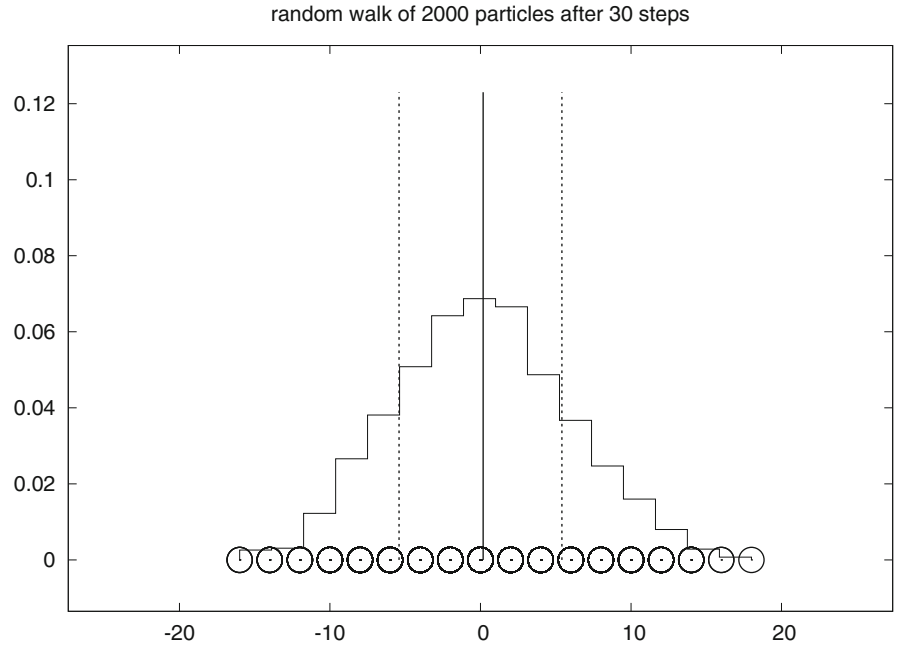

Fig. 8.6 Particle positions (circles), histogram (piecewise constant curve), and vertical lines indicating the mean value and the standard deviation from the mean after a one-dimensional random walk of 2000 particles for 30 steps

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

```
    mean_pos = numpy.mean(positions)
    stdev_pos = numpy.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), -0.01, 1.1*max(freq)],
     savefig='tmp.pdf')
```

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 = numpy.zeros(np)
    ypositions = numpy.zeros(np)
    # extent of the axis in the plot:
    xymax = 3*numpy.sqrt(ns); xymin = -xymax

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

    for step in range(ns):
        for i in range(np):
            direction = random.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=[xymin, xymax, xymin, xymax],
```

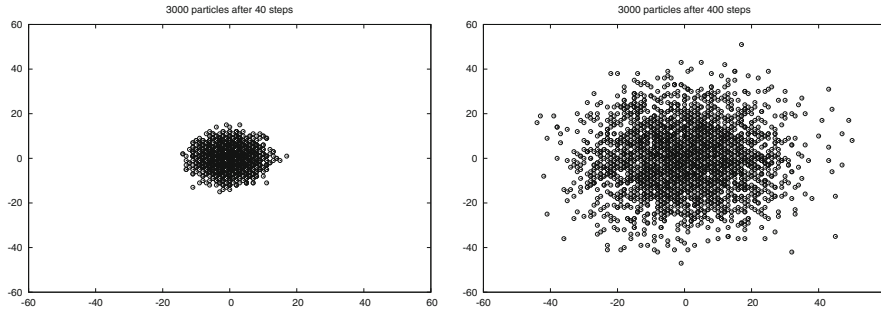

Fig. 8.7 Location of 3000 particles starting at the origin and performing a random walk, with 40 steps (*left*) and 400 steps (*right*)

```
                title='%d particles after %d steps' %
                      (np, step+1),
                savefig='tmp_%03d.pdf' % (step+1))
    return xpositions, ypositions

# main program:
import random
random.seed(10)
import sys
import numpy
from scitools.std import plot

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.

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 PDF files with names of the form `tmp_xxx.pdf`, where `xxx` is the step number. We can create a movie out of these individual files using the program `convert` from the ImageMagick suite:

Terminal

```
Terminal> convert -delay 50 -loop 1000 tmp_*.pdf 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. 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. The alternative is to make a true video file in, e.g., the Flash format:

Terminal

```
Terminal> avconv -r 5 -i tmp_%04d.png -c:v flv movie.flv
```

This requires the plot files to be in PNG format.

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. 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 += np.where(this_move == EAST, 1, 0)
xpositions -= np.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 = numpy.zeros(np)
    ypositions = numpy.zeros(np)
    moves = numpy.random.random_integers(1, 4, size=ns*np)
    moves.shape = (ns, np)

    # Estimate max and min positions
    xymax = 3*numpy.sqrt(ns); xymin = -xymax

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

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

```
        # Just plot every plot_step steps
        if (step+1) % plot_step == 0:
            plot(xpositions, ypositions, 'ko',
                 axis=[xymin, xymax, xymin, xymax],
                 title='%d particles after %d steps' %
                       (np, step+1),
                 savefig='tmp_%03d.pdf' % (step+1))
    return xpositions, ypositions

# Main program
from scitools.std import plot
import numpy, sys
numpy.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 (Sect. 8.1.2) and a normal (or Gaussian) distribution (Sect. 8.1.6).

The table below 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. Here, `N` is the array length in vectorized drawing, while `m` and `s` represent the mean and standard deviation values of a normal distribution. Functions from the standard `random` module appear in the middle column, while the corresponding functions from `numpy.random` are listed in the right column.

Functionality	Python's `random`	`numpy.random`
uniform numbers in $[0, 1)$	`random()`	`random(N)`
uniform numbers in $[a, b)$	`uniform(a, b)`	`uniform(a, b, N)`
integers in $[a, b]$	`randint(a, b)`	`randint(a, b+1, N)`
integers in $[a, b]$		`random_integers(a, b, N)`
Gaussian numbers	`gauss(m, s)`	`normal(m, s, N)`
set seed (`i`)	`seed(i)`	`seed(i)`
shuffle list in-place	`shuffle(a)`	`shuffle(a)`
choose a random element in list	`choice(a)`	

Typical probability computation via Monte Carlo simulation 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 (success):

```
import random
M = 0
for i in xrange(N):
    r = random.randint(a, b)
    if success:
        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 `success` 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

```
import numpy as np
r = np.random.uniform(a, b, N)
M = np.sum(condition)
# or
M = np.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 8.17.) Make sure you use `np.sum` 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 compute_histogram, plot
import numpy as np
m = np.mean(numbers)
v = np.var(numbers)
s = np.std(numbers)
x, y = compute_histogram(numbers, 50, piecewise_constant=True)
plot(x, y)
```

Terminology The important topics in this chapter are

- random numbers
- random number distribution
- Monte Carlo simulation
- Monte Carlo integration
- random walk

8.8.2 Example: Random Growth

Section A.1.1 presents simple mathematical models for how an investment grows when there is an interest rate being added to the investment at certain intervals. The models 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 γ 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 change in γ is postponed now).

Solution First we must develop the precise formulas to be implemented. The difference equations for x_n and p_n are simple in the present case, but the details of computing γ must be worked out.

In a program, we can draw two random numbers to estimate γ: 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 $1, \ldots, M$ 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 γ quantity:

$$x_n = x_{n-1} + \frac{p_{n-1}}{12 \cdot 100} x_{n-1}, \quad i = 1, \ldots, N \tag{8.11}$$

$$r_1 = \text{random integer in } [1, M] \tag{8.12}$$

$$r_2 = \text{random integer in } [1, 2] \tag{8.13}$$

$$\gamma = \begin{cases} m, & \text{if } r_1 = 1 \text{ and } r_2 = 1, \\ -m, & \text{if } r_1 = 1 \text{ and } r_2 = 2, \\ 0, & \text{if } r_1 \neq 1 \end{cases} \tag{8.14}$$

$$p_n = p_{n-1} + \begin{cases} \gamma, & \text{if } p_{n-1} + \gamma \in [1, 15], \\ 0, & \text{otherwise} \end{cases} \tag{8.15}$$

We remark that the evolution of p_n is much like a random walk process (Sect. 8.6), the only differences is that the plus/minus steps are taken at some random points among the times $0, 1, 2, \ldots, N$ rather than at all times $0, 1, 2, \ldots, N$. 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 $x_0, \ldots, x_n$ 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 = np.zeros(N+1)
    p = np.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.randint(1, M)
        if r == 1:
            # Adjust gamma
            r = random.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 the famous formula

$$x_n = x_0\left(1 + \frac{p}{12 \cdot 100}\right)^n$$

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 = np.zeros(N+1)
    pm = np.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 rounding 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 = np.zeros(N+1)  # standard deviation of x
    ps = np.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 = np.sqrt(xs)
    ps = np.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 wise 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`. This program applies time measurement methods from Sect. H.8.2.

Our `simulate_n_paths` function generates four arrays that 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); savefig('tmp_sample_paths_investment.pdf')
    figure(2); savefig('tmp_sample_paths_interestrate.pdf')
    ...
    return ...
```

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

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',
     savefig='tmp_mean_investment.pdf')
figure(4)
plot(months, pm, 'r',
     months, pm-ps, 'y',
     months, pm+ps, 'y',
     title='Mean +/- 1 st.dev. of annual interest rate',
     savefig='tmp_mean_interestrate.pdf')
```

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

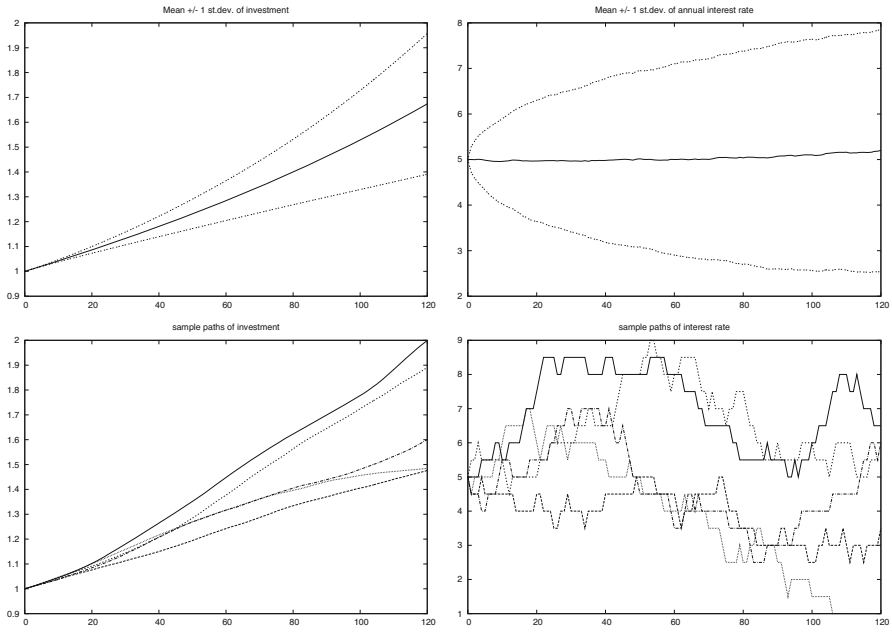

Fig. 8.8 Development of an investment with random jumps of the interest rate at random points of time. *Top left*: mean value of investment ± one standard deviation. *Top right*: mean value of the interest rate ± one standard deviation. *Bottom left*: five paths of the investment development. *Bottom right*: five paths of the interest rate development

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 Fig. 8.8.

8.9 Exercises

Exercise 8.1: Flip a coin times
Make a program that simulates flipping a coin N times. Print out `tail` or `head` for each flip and let the program count and print the number of heads.

Hint Use `r = random.random()` and define head as `r <= 0.5`, or draw an integer among $\{0, 1\}$ with `r = random.randint(0,1)` and define head when `r` is 0.
Filename: `flip_coin`.

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 $i = 1, 2, 3, 6$.
Filename: `compute_prob`.

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.
Filename: `choose_color`.

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?
Filename: `draw_10balls`.

Exercise 8.5: Computing probabilities of rolling dice
This exercise deals with four questions:

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.
Filename: `rolling_dice`.

Exercise 8.6: Estimate the probability in a dice game
Make a program for estimating the probability of getting at least one die with six eyes when throwing n dice. Read n and the number of experiments from the command line.

As a partial verification, compare the Monte Carlo simulation results to the exact answer 11/36 for $n = 2$ and observe that the approximate probabilities approach the exact probability as the number of simulations grow.
Filename: `one6_ndice`.

Exercise 8.7: Compute the probability of hands of cards
Use the `Deck.py` module (see Sect. 8.2.5) and the `same_rank` and `same_suit` functions from the `cards` module (see Sect. 8.2.4) 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.

Filename: `card_hands`.

Exercise 8.8: Decide if a dice game is fair
Somebody suggests the following game. You pay 1 euro and are allowed to throw four dice. If the sum of the eyes on the dice is less than 9, you get paid r euros back, otherwise you lose the 1 euro investment. Assume $r = 10$. Will you then in the long run win or lose money by playing this game? Answer the question by making a program that simulates the game. Read r and the number of experiments N from the command line.
Filename: `sum_4dice`.

Exercise 8.9: Adjust a game to make it fair
It turns out that the game in Exercise 8.8 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 Python function that computes the probability p of getting a sum less than s when rolling n dice. Use the reasoning in Sect. 8.3.2 to find the award per game, r, that makes the game fair. Run the program from Exercise 8.8 with this r on the command line and verify that the game is now (approximately) fair.
Filename: `sum_ndice_fair`.

Exercise 8.10: Make a test function for Monte Carlo simulation
We consider the Python function in Exercise 8.9 for computing a probability p that the sum of the eyes on n dice is less than s. The aim is to write a test function for verifying the computation of p.

a) Find some combinations of n and s that must result in $p = 0$ and $p = 1$ and make the appropriate code in the test function.
b) Fix the seed of the random number generator and record the first eight random numbers to 16 digits. Set $n = 2$, perform four experiments, and compute by hand what the probability estimate becomes (choose any appropriate s). Write the necessary code in the test function to compare this manually calculated result with the what is produced by the function from Exercise 8.9.

Filename: `test_sum_ndice`.

Exercise 8.11: Generalize a game
Consider the game in Sect. 8.3.2. A generalization is to think as follows: you throw one die until the number of eyes is less than or equal to the previous throw. Let m be the number of throws in a game.

a) Use Monte Carlo simulation to compute the probability of getting $m = 2, 3, 4, \ldots$.

Hint For $m \geq 6$ the throws must be exactly $1, 2, 3, 4, 5, 6, 6, 6, \ldots$, and the probability of each is 1/6, giving the total probability 6^{-m}. Use $N = 10^6$ experiments as this should suffice to estimate the probabilities for $m \leq 5$, and beyond that we have the analytical expression.

b) If you pay 1 euro to play this game, what is the fair amount to get paid when win? Answer this question for each of the cases $m = 2, 3, 4, 5$.

Filename: `incr_eyes`.

Exercise 8.12: Compare two playing strategies
Suggest a player strategy for the game in Sect. 8.4.2. Remove the question in the `player_guess` function in the file `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?
Filename: `simulate_strategies1`.

Exercise 8.13: Investigate strategies in a game
Extend the program from Exercise 8.12 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.
Filename: `simulate_strategies2`.

Exercise 8.14: Investigate the winning chances of some games
An amusement park offers the following game. A hat contains 20 balls: 5 red, 5 yellow, 3 green, and 7 brown. At a cost of $2n$ euros you can draw $4 \leq n \leq 10$ balls at random from the hat (without putting them back). Before you are allowed to look at the drawn balls, you must choose one of the following options:

1. win 60 euros if you have drawn exactly three red balls
2. win $7 + 5\sqrt{n}$ euros if you have drawn at least three brown balls
3. win $n^3 - 26$ euros if you have drawn exactly one yellow ball and one brown ball
4. win 23 euros if you have drawn at least one ball of each color

For each of the $4n$ different types of games you can play, compute the net income (per play) and the probability of winning. Is there any of the games (i.e., any combinations of n and the four options above) where you will win money in the long run?
Filename: `draw_balls`.

Exercise 8.15: Compute probabilities of throwing two dice
Throw two dice a large number of times in a program. Record the sum of the eyes each time and count how many times each of the possibilities for the sum (2, 3, . . ., 12) appear. Compute the corresponding probabilities and compare them with the exact values. (To find the exact probabilities, set up all the 6×6 possible outcomes of throwing two dice, and then count how many of them that has a sum s for $s = 2, 3, \ldots, 12$.)
Filename: `freq_2dice`.

Exercise 8.16: Vectorize flipping a coin
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 Constructions like `numpy.where(r<=0.5, 1, 0)` combined with `numpy.sum`, or `r[r<=0.5].size`, are useful, where `r` is an array of random numbers between 0 and 1.
Filename: `flip_coin_vec`.

Exercise 8.17: Vectorize a probablility computation
The purpose of this exercise is to speed up the code in Exercise 8.2 by vectorization.

Hint For an array `r` of uniformly distributed random numbers on [0, 1), make use of `r1 = r[r>0.5]` and `r1[r1<0.6]`. An alternative is `numpy.where` combine with a compound boolean expression with `numpy.logical_and(0.5>=r, r<=0.6)`. See the discussion of this topic in Sect. 5.5.3.
Filename: `compute_prob_vec`.

Exercise 8.18: Throw dice and compute a small probability
Use Monte Carlo simulation to compute the probability of getting 6 eyes on all dice when rolling 7 dice.

Hint You need a large number of experiments in this case because of the small probability (see the first paragraph of Sect. 8.3), so a vectorized implementation may be important.
Filename: `roll_7dice`.

Exercise 8.19: Is democracy reliable as a decision maker?
A democracy takes decisions based on majority votes. We shall investigate if this is a good idea or if a single person would produce better decisions.

We shall ask about pure facts, not opinions. This means that the question to be answered by a population has a definite "yes" or "no" answer. For example, "Can Python lists contain tuples as elements?" The correct answer is "yes". Asking a population such a question and relying on the majority of votes, is a reliable procedure if the competence level in the population is sufficiently high.

a) Assume that the competence level in a population can be modeled by a probability p such that if you ask N people a question, $M = pN$ of them will give

the correct answer (as $N \to \infty$). Here we make the questionable assumption of a homogeneous population, in the sense that p is the same for every individual. Make a function `homogeneous(p, N)` for simulating whether the majority vote of a population of N individuals arrives at the right answer, if the probability of answering correctly is p for an individual. Make another function `homogeneous_ex()` that runs 10 tests the specific case of $N = 5$ (as when relying on the majority of a student group) and 10 tests when asking a whole city of $N = 1,000,000$ voters. Try $p = 0.49$, $p = 0.51$, and $p = 0.8$. Are the results as you would expect from intuition?

Hint Asking one individual is like flipping a biased coin that has probability p of giving head (right answer) and probability $1 - p$ of giving tail (wrong answer).

b) The problem in a) can be exactly solved, since each question is a Bernoulli trial with success probability p, and the probability of a correct majority vote is the same as the probability of getting $N/2$ *or more* successes in N trials. For large N, the probability of M successes in N trials can be well approximated by a normal (Gaussian) density function:

$$g(M) = (\sqrt{2\pi}Np(1-p))^{-1} \exp\left(-\frac{1}{2}(M - Np)^2/(Np(1-p))\right).$$

The majority vote is correct when $M > N/2$, and the probability of this event is given by $1 - \Phi(N/2)$, where Φ is the cumulative normal distribution with mean Np and variance $Np(1-p)$.
Plot the probability of being right against p.
Say 5 questions are of importance. What competence level p does a king need to have all 5 right compared to the population having all 5 right.

c) We shall now simulate voting in a *heterogeneous* population. The probability that an individual no. i answers correctly is p_i, where p_i is drawn from a normal (Gaussian) probability density with mean p and standard deviation s. The competence level varies between individuals, with s expressing the spreading of knowledge and p the mean competence level.
Make function `heterogeneous(p, N, s)` for returning whether the majority vote is right or wrong in the heterogeneous case. Rerun the examples from a) with $s = 0.2$.

d) With a somewhat large variation of the population, i.e., s somewhat large, there will be some individuals that always provide wrong or right answers according to this model. To learn about reasonable values s we can investigate unreasonable large amounts of people who are *always* right or wrong.
The probability of always being wrong is the probability of $p_i < 0$. This is given by $\Phi(-p/s)$, where Φ is the cumulative normal distribution with mean zero and unit standard deviation. It can be reached in Python as `scipy.stats.norm.cdf`. The probability of always being right is the probability of $p_i > 1$, which can be computed as $1 - \Phi((1-p)/s)$. Plot curves of the probability of always being right and always wrong against $s \in [0.1, 0.6]$. Perform this curve plotting in a function `extremes(p)`.

Filename: `democracy`.

Exercise 8.20: Difference equation for random numbers
Simple random number generators are based on simulating difference equations. Here is a typical set of two equations:

$$x_n = (ax_{n-1} + c) \bmod m, \tag{8.16}$$
$$y_n = x_n/m, \tag{8.17}$$

for $n = 1, 2, \ldots$. A seed x_0 must be given to start the sequence. The numbers $y_1, y_2, \ldots$ represent the random numbers and $x_0, x_1, \ldots$ are "help" numbers. Although y_n is completely deterministic from (8.16)–(8.17), the sequence y_n *appears* random. The mathematical expression p mod q is coded as `p % q` in Python.

Use $a = 8121$, $c = 28411$, and $m = 134456$. Solve the system (8.16)–(8.17) in a function to generate N random numbers. Make a histogram to examine the distribution of the numbers (the y_n numbers are uniformly distributed if the histogram is approximately flat).
Filename: `diffeq_random`.

Exercise 8.21: Make a class for drawing balls from a hat
Consider the example about drawing colored balls from a hat in Sect. 8.3.3. It could be handy to have an object that acts as a hat:

```
hat = Hat(red=3, blue=4, green=6)
balls = hat.draw(3)
if balls.count('red') == 1 and balls.count('green') == 2:
    ...
```

a) Write such a class `Hat` with the shown functionality.

Hint 1 The flexible syntax in the constructor, where the colors of the balls and the number of balls of each color are freely specified, requires use of a dictionary (`**kwargs`) for handling a variable number of keyword arguments, see Sect. H.7.2.

Hint 2 You can borrow useful code from the `balls_in_hat.py` program and ideas from Sect. 8.2.5.

b) Apply class `Hat` to compute the probability of getting 2 brown and 2 blue galls when drawing 6 balls from a hat with 6 blue, 8 brown, and 3 green balls.

Filename: `Hat`.

Exercise 8.22: Independent versus dependent random numbers

a) 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 `0010110101101110 10`).
b) The purpose now 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.

c) Choose $N = 80$ and try the probabilities $p = 0.5$, $p = 0.8$ and $p = 0.9$. Can you by visual inspection of the output characterize the differences between sequences of independent and dependent random variables?

Filename: `dependent_random_numbers`.

Exercise 8.23: Compute the probability of flipping a coin

a) Simulate flipping a coin N times.

Hint Draw N random integers 0 and 1 using `numpy.random.randint`.

b) Look at a subset $N_1 \leq N$ of the experiments in a) and compute the 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?

c) Now we want to study the probability of getting a head, p, as a function of N_1, i.e., for $N_1 = 1, \ldots, N$. A first try to compute the probability array for p is

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

Implement these computations in a function.

d) An array `q[i] = np.sum(h([:i]))` reflects a *cumulative sum* and can be efficiently generated by `np.cumsum`: `q = np.cumsum(h)`. Thereafter we can compute `p` by `q/I`, where `I[i]=i+1` and `I` can be computed by `np.arange(1,N+1)` or `r_[1:N+1]` (integers 1, 2, ..., up to but not including `N+1`). Use `cumsum` to make an alternative vectorized version of the function in c).

e) Write a test function that verifies that the implementations in c) and d) give the same results.

Hint Use `numpy.allclose` to compare two arrays.

f) Make a function that applies the `time` module to measure the relative efficiency of the implementations in c) and d).

g) Plot `p` against `I` for the case where $N = 10000$. Annotate the axis and the plot with relevant text.

Filename: `flip_coin_prob`.

Exercise 8.24: Simulate binomial experiments
Exercise 4.24 describes some problems that can be solved exactly using the formula (4.8), 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 (4.8)). Simulate the three cases in Exercise 4.24 using this function.
Filename: `simulate_binomial`.

Exercise 8.25: Simulate a poker game
Make a program for simulating the development of a poker (or simplified poker) game among n players. Use ideas from Sect. 8.2.4.
Filename: `poker`.

Exercise 8.26: Estimate growth in a simulation model
The simulation model in Sect. 8.3.5 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 a fraction 0.06 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 as n increases.
Filename: `estimate_growth`.

Exercise 8.27: Investigate guessing strategies
In the game from Sect. 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 one of the strategies can be considered as superior in the long run.
Filename: `strategies4guess`.

Exercise 8.28: Vectorize a dice game
Vectorize the simulation program from Exercise 8.8 with the aid of the module `numpy.random` and the `numpy.sum` function.
Filename: `sum9_4dice_vec`.

Exercise 8.29: Compute π by a Monte Carlo method
Use the method in Sect. 8.5.3 to compute π 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 π with `math.pi`.
Filename: `MC_pi`.

Exercise 8.30: Compute π by a Monte Carlo method
This exercise has the same purpose of computing π as in Exercise 8.29, 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 $(x - x_c)^2 + (y - y_c)^2 < R^2$.
Filename: `MC_pi2`.

Exercise 8.31: Compute π by a random sum

a) Let $x_0, \ldots, x_N$ be $N + 1$ uniformly distributed random numbers between 0 and 1. Explain why the random sum $S_N = (N + 1)^{-1} \sum_{i=0}^{N} 2(1 - x_i^2)^{-1/2}$ is an approximation to π.

Hint Interpret the sum as Monte Carlo integration and compute the corresponding integral by hand or `sympy`.

b) Compute $S_0, S_1, \ldots, S_N$ (using just one set of $N + 1$ random numbers). Plot this sequence versus N. Also plot the horizontal line corresponding to the value of π. Choose N large, e.g., $N = 10^6$.

Filename: `MC_pi_plot`.

Exercise 8.32: 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 $n_p \to \infty$ (n_s is the number of walks). Write out this exact result together with the computed mean position with a finite number of particles.
Filename: `walk1D_drift`.

Exercise 8.33: 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 $x_p = 5, 50, 5000, 50000$.
Filename: `walk1Dv_hit_point`.

Exercise 8.34: Simulate making a fortune from gaming
A man plays a game where the probability of winning is p and that of losing is consequently $1 - p$. When winning he earns 1 euro and when losing he loses 1 euro. Let x_i be the man's fortune from playing this game i number of times. The starting fortune is x_0. We assume that the man gets a necessary loan if $x_i < 0$ such that the gaming can continue. The target is a fortune F, meaning that the playing stops when $x = F$ is reached.

a) Explain why x_i is a 1D random walk.
b) Modify one of the 1D random walk programs to simulate the average number of games it takes to reach the target fortune $x = F$. This average must be computed by running a large number of random walks that start at x_0 and reach F. Use $x_0 = 10$, $F = 100$, and $p = 0.49$ as example.
c) Suppose the average number of games to reach $x = F$ is proportional to $(F - x_0)^r$, where r is some exponent. Try to find r by experimenting with the program. The r value indicates how difficult it is to make a substantial fortune by playing this game. Note that the *expected* earning is negative when $p < 0.5$, but there is still a small probability for hitting $x = F$.

Filename: `game_as_walk1D`.

Exercise 8.35: Simulate pollen movements as a 2D random walk
The motion of single particles can often be described as random walks. On a water surface, 1000 grains of pollen are placed in a single point. The movement of the pollen grains can be modeled by a random walk model, where for each second each grain will move a random distance, along a two-dimensional vector, whose two components are independently normally distributed with expectation 0 mm and standard deviation 0.05 mm.

a) Make a function that implements this kind of 2D random walk. Return an array with the position of each grain for each step.
b) Make a movie that shows the position of the pollen grains from 0 to 100 seconds.
c) Make a plot of the mean distance from the origin versus time. What do you see?

Filename: `pollen`.

Exercise 8.36: Make classes for 2D random walk
The purpose of this exercise is to reimplement the `walk2D.py` program from Sect. 8.7.1 with the aid of classes.

a) Make a class `Particle` with the coordinates (x, y) and the time step number of a particle as data 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.

b) 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 so the output looks nice.

c) Make a test function that compares the first three positions of four particles with the corresponding results computed by the `walk2D.py` program. The seed of the random number generator must of course be fixed identically in the two programs.
d) Organize the complete code as a module such that the classes `Particle` and `Particles` can be reused in other programs. The test block should read the number of particles from the command line and perform a simulation.
e) Compare the efficiency of the class version against the vectorized version in `walk2Dv.py`, using the techniques in Sect. H.8.1.
f) The program developed above 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 make a new class `Particles_vec` which vectorizes `Particles`.
g) Verify the code against the `walk2Dv.py` program as explained in c). Automate the verification in a test function.
h) Write a Python function that measures the computational efficiency the vectorized class `Particles_vec` and the scalar class `Particles`.

Filename: `walk2D_class`.

Exercise 8.37: 2D random walk with walls; scalar version
Modify the `walk2D.py` or `walk2Dc.py` programs from Exercise 8.36 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 its new position is outside A.
Filename: `walk2D_barrier`.

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 and move them back to the relevant boundary of A.
Filename: `walk2Dv_barrier`.

Exercise 8.39: Simulate mixing 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?
Filename: `disorder1`.

Remarks 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.

Exercise 8.40: Simulate slow mixing of gas molecules
Solve Exercise 8.39 when the wall dividing the box is not completely removed, but instead has a small hole.
Filename: `disorder2`.

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 $b = 1/2$?
Filename: `simulate_beer_tasting`.

Exercise 8.42: Simulate stock prices
A common mathematical model for the evolution of stock prices can be formulated as a difference equation

$$x_n = x_{n-1} + \Delta t \mu x_{n-1} + \sigma x_{n-1}\sqrt{\Delta t} r_{n-1}, \tag{8.18}$$

where x_n is the stock price at time t_n, Δt is the time interval between two time levels ($\Delta t = t_n - t_{n-1}$), μ is the growth rate of the stock price, σ is the volatility of the stock price, and $r_0, \ldots, r_{n-1}$ 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 μ, σ, and Δt.

We can make a remark that (8.18) is a Forward Euler discretization of a stochastic differential equation for a continuous price function $x(t)$:

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

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 (8.18) for $n = 0, \ldots, N$ for $N = 5000$ steps over a time period of $T = 180$ days with a step size $\Delta t = T/N$.
Filename: `stock_prices`.

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,

$$S(t+1) = (1+r)S(t) + \sigma S(t)\epsilon(t), \tag{8.19}$$

where r is the interest-rate, and σ is the volatility of the stock price. The time t is supposed to be measured in days, $t = 0, 1, 2, \ldots$, 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

$$p = (1+r)^{-T}\mathrm{E}\left[\max\left(\frac{1}{T}\sum_{t=1}^{T} S(t) - K, 0\right)\right]. \tag{8.20}$$

The price is thus given as the expected discounted payoff. We will use Monte Carlo simulations to estimate the expectation. Typically, r and σ 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 $t = 1, \ldots, T$ for a given T based on the recursive definition in (8.19). 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 $1/\sqrt{N}$.

Filename: `option_price`.

Remarks If you wonder where the values for r and σ come from, you will find the explanation in the following. A reasonable level for the yearly interest-rate is around 5 percent, 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 σ 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 percent 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 (8.19) must be substituted with μ, the expected return of the stock. Usually, μ is higher than r.

Exercise 8.44: Differentiate noise measurements
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 η, 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: $\eta(t_0), \eta(t_1), \eta(t_2), \ldots, \eta(t_n)$. There are 300 observations per second, meaning that the time difference between two neighboring measurement values $\eta(t_i)$ and $\eta(t_{i+1})$ is $h = 1/300$ second.

a) Read the η values in the file `gauge.dat`[4] into an array `eta`. Read h from the command line.
b) Plot `eta` versus the time values.
c) Compute the velocity v of the surface by the formula

$$v_i \approx (\eta_{i+1} - \eta_{i-1})/(2h), \quad i = 1, \ldots, n-1\,.$$

Plot v versus time values in a separate plot.

[4] http://tinyurl.com/pwyasaa/random/gauge.dat

d) Compute the acceleration a of the surface by the formula

$$a_i \approx (\eta_{i+1} - 2\eta_i + \eta_{i-1})/h^2, \quad i = 1, \ldots, n-1.$$

Plot a versus the time values in a separate plot.

e) If we have a noisy signal η_i, where $i = 0, \ldots, n$ 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 η_i, $i = 0, \ldots, n$, we compute a filtered (averaged) signal with values $\eta_i^{(1)}$ by the formula

$$\eta_i^{(1)} = \frac{1}{4}(\eta_{i+1} + 2\eta_i + \eta_{i-1}), \quad i = 1, \ldots, n-1, \ \eta_0^{(1)} = \eta_0, \ \eta_n^{(1)} = \eta_n \,. \tag{8.21}$$

Make a function `filter` that takes the η_i values in an array `eta` as input and returns the filtered $\eta_i^{(1)}$ values in an array.

f) Let $\eta_i^{(k)}$ be the signal arising by applying the `filtered` function k times to the same signal. Make a plot with curves η_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, measured η data and the filtered data. Discuss the results.

Filename: `labstunami`.

Exercise 8.45: Differentiate 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 Exercise 8.44.

a) Compute a signal

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

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`.

b) 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 η_i signal as circles in the same plot as $\bar{\eta}_i$. Store the E_i in an array `E` for later use.

c) Compute the first derivative of $\bar{\eta}_i$ by the formula

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

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

d) Compute the first derivative of the error term by the formula

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

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

e) Plot `detabar` and `detabar + dE`. Use the result of the standard deviation calculations to explain the qualitative features of the graphs.

f) The second derivative of a time signal η_i can be computed by

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

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.

Filename: `sine_noise`.

Exercise 8.46: Model noise in a time signal

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.

a) 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 σ. 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 σ, magnified when we use this numerical approximation of the velocity?

b) 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.

c) The numbers in the `gauge.dat` file in Exercise 8.44 are given with 5 digits. This is no certain indication of the accuracy of the measurements, but as a test we may assume σ 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.

Exercise 8.47: Speed up Markov chain mutation
The functions `transition` and `mutate_via_markov_chain` from Sect. 8.3.4 were made for being easy to read and understand. Upon closer inspection, we realize that the `transition` function constructs the `interval_limits` every time a random transition is to be computed, and we want to run a large number of transitions. By merging the two functions, pre-computing interval limits for each `from_base`, and adding a loop over `N` mutations, one can reduce the computation of interval limits to a minimum. Perform such an efficiency enhancement. Measure the CPU time of this new function versus the `mutate_via_markov_chain` function for 1 million mutations.
Filename: `markov_chain_mutation2`.