out of the question, but we shall see that it is sometimes possible to squeeze all configurations in between two extremal ones: if those two configurations have come together, all others have merged, too.

Understanding and running a coupling-from-the-past program is the ultimate in Monte Carlo style—much more elegant than walking around a heliport, well dressed and with a fancy handbag over one's shoulder, waiting for the memory of the clubhouse to more or less fade away.

1.2 Basic sampling

On several occasions already, we have informally used uniform random numbers x generated through a call $x \leftarrow \texttt{ran}\,(a, b)$. We now discuss the two principal aspects of random numbers. First we must understand how random numbers enter a computer, a fundamentally deterministic machine. In this first part, we need little operational understanding, as we shall always use routines written by experts in the field. We merely have to be aware of what can go wrong with those routines. Second, we shall learn how to reshape the basic building block of randomness—$\texttt{ran}\,(0, 1)$—into various distributions of random integers and real numbers, permutations and combinations, N-dimensional random vectors, random coordinate systems, etc. Later chapters will take this program much further: $\texttt{ran}\,(0, 1)$ will be remodeled into random configurations of liquids and solids, boson condensates, and mixtures, among other things.

1.2.1 Real random numbers

Random number generators (more precisely, pseudorandom number generators), the subroutines which produce $\texttt{ran}\,(0, 1)$, are intricate deterministic algorithms that condense into a few dozen lines a lot of clever number theory and probabilities, all rigorously tested. The output of these algorithms looks random, but is not: when run a second time, under exactly the same initial conditions, they always produce an identical output. Generators that run in the same way on different computers are called "portable". They are generally to be preferred. Random numbers have many uses besides Monte Carlo calculations, and rely on a solid theoretical and empirical basis. Routines are widely available, and their writing is a mature branch of science. Progress has been fostered by essential commercial applications in coding and cryptography. We certainly do not have to conceive such algorithms ourselves and, in essence, only need to understand how to test them for our specific applications.

Every modern, good $\texttt{ran}\,(0, 1)$ routine has a flat probability distribution. It has passed a battery of standard statistical tests which would have detected unusual correlations between certain values $\{x_i, \ldots, x_{i+k}\}$ and other values $\{x_j, \ldots, x_{j+k'}\}$ later down the sequence. Last but not least, the standard routines have been successfully used by many people before us. However, all the meticulous care taken and all the endorsement by others do not insure us against the small risk that the particular random number generator we are using may in fact fail in our particular

problem. To truly convince ourselves of the quality of a complicated calculation that uses a given random number generator, it remains for us (as end users) to replace the random number generator *in the very simulation program we are using* by a second, different algorithm. By the definition of what constitutes randomness, this change of routine should have no influence on the results (inside the error bars). Therefore, if changing the random number generator in our simulation program leads to no systematic variations, then the two generators are almost certainly OK for our application. There is nothing more we can do and nothing less we should do to calm our anxiety about this crucial ingredient of Monte Carlo programs.

Algorithm 1.9 (`naive-ran`) is a simple example—useful for study, but unsuited for research—of linear congruential[3] random number generators, which are widely installed in computers, pocket calculators, and other digital devices. Very often, such generators are the building blocks of good algorithms.

Table 1.6 Repeated calls to Alg. 1.9 (`naive-ran`). Initially, the seed was set to idum ← 89053.

#	idum	ran
1	123456	0.91819
2	110651	0.82295
3	55734	0.41451
4	65329	0.48588
5	1844	0.01371
6	78919	0.58695
...	...	...
134457	123456	...
134458	110651	...
...	...	...

procedure `naive-ran`
$m \leftarrow 134456$
$n \leftarrow 8121$
$k \leftarrow 28411$
input idum
idum $\leftarrow$ `mod`(idum $\cdot\, n + k, m$)
`ran` $\leftarrow$ idum/`real`(m)
output idum, `ran`

Algorithm 1.9 `naive-ran`. Low-quality portable random number generator, `naive-ran`$(0, 1)$, using a linear congruential method.

In Alg. 1.9 (`naive-ran`), the parameters $\{m, n, k\}$ have been carefully adjusted, whereas the variable {idum}, called the seed, is set at the beginning, but never touched again from the outside during a run of the program. Once started, the sequence of pseudorandom numbers unravels. Just like the sequence of any other generator, even the high-quality ones, it is periodic. In Alg. 1.9 (`naive-ran`), the periodicity is 134 456 (see Table 1.6); in good generators, the periodicity is much larger than we shall ever be able to observe.

Let us denote real random numbers, uniformly distributed between values a and b, by the abstract symbol `ran`$\,(a, b)$, without regard for initialization and the choice of algorithm (we suppose it to be perfect). In the printed routines, repeated calls to `ran`$\,(a, b)$, such as

$$\begin{aligned} x &\leftarrow \texttt{ran}\,(-1, 1)\,, \\ y &\leftarrow \texttt{ran}\,(-1, 1)\,, \end{aligned} \tag{1.19}$$

generate statistically independent random values for x and y. Later, we shall often use a concise vector notation in our programs. The two

[3]Two numbers are *congruent* if they agree with each other, i.e. if their difference is divisible by a given modulus: 12 is congruent to 2 (modulo 5), since $12 - 2 = 2 \times 5$.

variables $\{x, y\}$ in eqn (1.19), for example, may be part of a vector $\mathbf{x}$, and we may assign independent random values to the components of this vector by the call

$$\mathbf{x} \leftarrow \{\mathtt{ran}\,(-1,1)\,, \mathtt{ran}\,(-1,1)\}.$$

(For a discussion of possible conflicts between vectors and random numbers, see Subsection 1.2.6.)

Depending on the context, random numbers may need a little care. For example, the logarithm of a random number between 0 and 1, $x \leftarrow \log \mathtt{ran}\,(0,1)$, may have to be replaced by

```
1  ϒ ← ran(0,1)
   if (ϒ = 0) goto 1 (reject number)
   x ← log ϒ
```

to avoid overflow ($\Upsilon = 0$, $x = -\infty$) and a crash of the program after a few hours of running. To avoid this problem, we might define the random number $\mathtt{ran}\,(0,1)$ to be always larger than 0 and smaller than 1. However this does not get us out of trouble: a well-implemented random number generator between 0 and 1, always satisfying

$$0 < \mathtt{ran}\,(0,1) < 1,$$

might be used to implement a routine $\mathtt{ran}\,(1,2)$. The errors of finite-precision arithmetic could lead to an inconsistent implementation where, owing to rounding, $1 + \mathtt{ran}\,(0,1)$ could turn out to be exactly equal to one, even though we want $\mathtt{ran}\,(1,2)$ to satisfy

$$1 < \mathtt{ran}\,(1,2) < 2.$$

Clearly, great care is called for, in this special case but also in general: Monte Carlo programs, notwithstanding their random nature, are extremely sensitive to small bugs and irregularities. They have to be meticulously written: under no circumstance should we accept routines that need an occasional manual restart after a crash, or that sometimes produce data which has to be eliminated by hand. Rare problems, for example logarithms of zero or random numbers that are equal to 1 but should be strictly larger, quickly get out of control, lead to a loss of trust in the output, and, in short, leave us with a big mess ....

1.2.2 Random integers, permutations, and combinations

Random variables in a Monte Carlo calculation are not necessarily real-valued. Very often, we need uniformly distributed random integers m, between (and including) k and l. In this book, such a random integer is generated by the call $m \leftarrow \mathtt{nran}\,(k,l)$. In the implementation in Alg. 1.10 (`nran`), the **if ( )** statement (on line 4) provides extra protection against rounding problems in the underlying $\mathtt{ran}\,(k, l+1)$ routine.