The functions `srand` and `rand` allow programs to generate a stream of apparently uncorrelated integers. The sequence is completely determined by the seed passed to `srand`. Care should be taken that the stream of values generated is indeed suited to the application they are required for.

The simulation carried out by the program in the top box in Figure 9.5 shows how one particular participant gets on in the hypothetical game of chance. However the casino is interested in the long term statistical behavior of the game, not individual players. The third box of the figure shows the output of a modified program, one which, starting with the same seed and then never reinitializing it, commences the process with a \$5 initial stake $100{,}000$ times. A computer-based rent-a-crowd, if you like, only considerably cheaper. Now the reason the casino likes this game is obvious – about $6/7$ of the players eventually lose their stake, and only $1/7$ of players ever make it to \$20.

The ready ability of computers to undertake brute-force computations means that it is also easy to check the stability of such a numerical result, by running the entire program repeatedly, with different seeds each time. For example, when `SEED` is initialized to $12345678$, the same program records $13{,}712$ winners and $86{,}288$ losers, and the averages are $67.9$ and $34.8$ games respectively. Similar results from other seeds suggest that the values in the bottom box of Figure 9.5 are reasonably precise.

The simulation strategy is appropriate when a large amount of randomly generated data can be collated to predict a meaningful overall trend. Multiple runs should be undertaken in order to verify the stability of the answers.

The simulation shown in this example is based upon a single stream of actions. In more general situations each event that is processed may cause some number of future events to be scheduled. For example, in a program modeling the average queue length in a multi-teller bank branch, the arrival of a customer at a teller decreases the length of the queue by one, and requires that a future "customer departs teller" event be scheduled, with the delay between the two events a function of the complexity of the transaction assigned to that customer by the simulation. A queue of pending events is maintained, and processed in simulation-time order.

Discrete event simulations of this kind can become rather complex. One thing worth remembering when using the output of such a program to model real-world behavior is that the projections made by the simulation are valid only if the input assumptions governing the model are realistic. In a simulation of a bank branch, for example, allowing the queue of waiting customers to become arbitrarily long is unrealistic, as real customers entering a real branch will simply turn around and leave again if they see a long queue and their banking business is non-urgent.

Randomness can also be used to help find solutions to other problems. Suppose some complex geometric shape is given, and the internal area of it is required. For example, Figure 9.7 shows an arc-shaped region defined by a unit circle. In this simple example the area can be directly calculated as $\pi/4 - 0.5 = 0.2854$. But suppose that the shape was more complex, and standard geometric formulae could

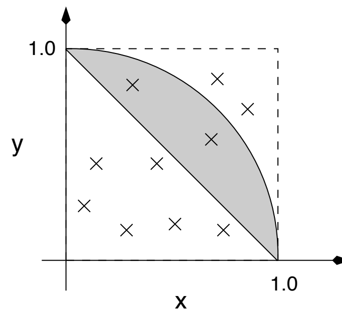


**Figure 9.7:** Monte Carlo estimation of Figure 9.8 with `steps` equal to ten.

not be used. Suppose also that a mechanism is known for determining if a point lies inside or outside the region of interest. In Figure 9.7, points $(x, y)$ inside the grey region satisfy $(1 - x) \leq y$ and $x^2 + y^2 \leq 1$.

Figure 9.8 shows a program that estimates the area of the shaded region, based purely upon the inside/outside test. Random $(x, y)$ locations are generated using `rand`, and checked against the region. The small "x"s in Figure 9.7 are perhaps the

```
int
inside(double x, double y) {
    return ((1-x)<=y && x*x+y*y<=1.0);
}
```

```
    srand(SEED);
    for (steps=1; steps <= 1000000; steps=steps*10) {
        num_in=0;
        for (i=0; i<steps; i++) {
            x = rand()/(1.0+RAND_MAX);
            y = rand()/(1.0+RAND_MAX);
            num_in = num_in + inside(x,y);
        }
        printf("steps = %7d, num_in = %8d, ratio = %.6f\n",
            steps, num_in, (double)num_in/steps);
    }
```

```
steps =       1, num_in =        0, ratio = 0.000000
steps =      10, num_in =        2, ratio = 0.200000
steps =     100, num_in =       28, ratio = 0.280000
steps =    1000, num_in =      287, ratio = 0.287000
steps =   10000, num_in =     2896, ratio = 0.289600
steps =  100000, num_in =    28514, ratio = 0.285140
steps = 1000000, num_in =   285728, ratio = 0.285728
```

**Figure 9.8:** Monte Carlo estimation of the area contained in an arc-shaped region of a unit circle. The top box shows a function that returns true if the $(x, y)$ location is inside the region of interest. The third box shows an execution of the program fragment in the middle box.