AMTH142 Lecture 14

Monte-Carlo Integration
Simulation

April 20, 2007

All the Scilab functions defined in this Lecture can be found in the file `l14.sci` in the directory for this lecture.

Contents

14.1 Monte-Carlo Integration 2
  14.1.1 One Dimension 2
  14.1.2 The Central Limit Theorem 4
  14.1.3 Estimating Volumes 5
  14.1.4 Multiple Integrals 6
14.2 Simulation 9
  14.2.1 Two Dice 9
  14.2.2 Errors 10
  14.2.3 Two Loaded Dice 12
  14.2.4 The Birthday Problem 13
14.3 Appendix – Boolean Matrices 14
  14.3.1 Comparison and Logical Operators 14
  14.3.2 Real and Boolean Matrices 15
  14.3.3 `find` 16

14.1 Monte-Carlo Integration

14.1.1 One Dimension

We wish to evaluate an integral

$$I = \int_a^b f(x)dx$$

The average value of $f(x)$ is

$$\bar{f} = \frac{1}{b-a} I$$

so that

$$I = (b-a)\bar{f}$$

Suppose we choose points $x_1, \ldots, x_n$ randomly in the interval $[a, b]$ and use these to estimate the average value of $f(x)$:

$$\bar{f} \approx \frac{1}{n} \sum_{i=1}^{n} f(x_i)$$

then

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

This is Monte-Carlo integration.

Implementation

Here is a Scilab implementation of Monte-Carlo integration of a function `f` over the interval `[a,b]` using `n` points.

```
function ii = monte1d(f, a, b, n)
  x = (b-a)*rand(1,n) + a   // choose n random points in [a,b]
  fx = f(x)                 // evaluate f(x) at each point
  ii = (b-a)*sum(fx)/n      // (b-a) * average of function
endfunction
```

Example

Let us estimate the integral

$$I = \int_0^{2\pi} e^{-x} \sin(x) dx$$

The exact value, from integration by parts, is

$$I = \frac{1}{2}(1 - e^{-2\pi}) = 0.4990663.$$

Here is a Monte-Carlo estimate using Scilab. The first step is to define the function we want to integrate

```
-->function y = f(x)
-->  y = exp(-x).*sin(x)
-->endfunction

-->ii = monte1d(f, 0, 2*%pi, 100000)
 ii  =

    0.4983886
```

We can compare to the exact answer and find the relative error:

```
-->ie = (1 - exp(-2*%pi))/2
 ie  =

    0.4990663

-->err = abs(ii - ie)/ie
 err  =

    0.0013579
```

i.e. with 10,000 points we get a relative error of the order of 0.1%.

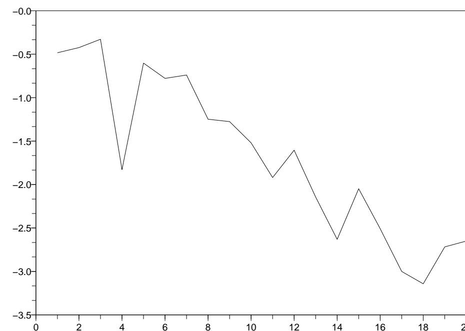
Let us see how the error varies with the number of points `n`. We will perform a Monte-Carlo approximation with $2, 4, 8, \ldots, 2^{20}$ points using a simple `for` loop and then compute the error for each approximation:

```
-->in = zeros(1,20);

-->for k = 1:20
-->  in(k) = monte1d(f, 0 , 2*%pi, 2^k);
-->end

-->err = abs(in - ie)/ie
 err  =


         column 1 to 4

!   0.3288851    0.3764572    0.4697016    0.0148549 !

         column 5 to 8

!   0.2502828    0.1665171    0.1823111    0.0565038 !

         column  9 to 12
```

```
!   0.0530368    0.0301861    0.0120509    0.0248556 !

         column 13 to 16

!   0.0071772    0.0023424    0.0089532    0.0031000 !

         column 17 to 20

!   0.0009980    0.0007203    0.0019182    0.0022399 !
```

As expected the error decreases as the number of points increases. Plotting the log of the error is the most instructive:

```
-->plot2d(log10(err))
```



14.1.2 The Central Limit Theorem

The central limit theorem of probability theory gives an estimate of the error in Monte-Carlo integration. For our purposes it can be formulated as follows: Suppose the mean of a function $f(x)$ is estimated by random sampling

$$\bar{f}_{\text{est}} = \frac{1}{n}\sum_{i=1}^{n} f(x_i)$$

Then the variance of the estimated mean is

$$\text{Var}\,\bar{f} = \frac{\sigma^2}{n}$$

where σ^2 is the variance of $f(x)$.

It is usual to take the standard deviation as a measure of error. Then we have

$$\text{Error} = \frac{\sigma}{\sqrt{n}}$$

The important point here is that the error goes to zero like $1/\sqrt{n}$. This says, for example, that to decrease the error by a factor of 1000, we must increase the sample size by a factor of 1000000.

We will compare our results in the previous example with this theory. The variance of $f(x) = e^{-x}\sin x$ is

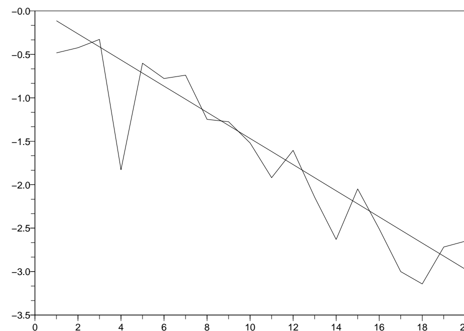
$$\sigma^2 = \int_0^{2\pi} (f(x) - \bar{f})^2 \approx 1.19$$

so the expected error in Monte-Carlo integration is

$$E_n = \frac{\sigma}{\sqrt{n}} \approx \frac{1.09}{\sqrt{n}}$$

We can create a semi-log plot of the estimated Monte-Carlo error as follows:

```
-->n = 1:20;

-->ee = 1.09*(2 .^n).^(-1/2);

-->plot2d(log10(ee))
```



The error in the computed values show random fluctuations, as is to be expected, but that the theory gives a good account of the error.

14.1.3 Estimating Volumes

What is the volume of a unit sphere in 4-dimensions? We can obtain an estimate by Monte-Carlo methods.

The unit sphere is the region

$$x_1^2 + x_2^2 + x_3^2 + x_4^2 \leq 1$$

If we generate random points in the four dimensional cubic region $[-1,1]\times[-1,1]\times[-1,1]\times[-1,1]$ which contains the unit sphere, then an estimate of the volume of the sphere

$$\frac{\text{Volume of Sphere}}{\text{Volume of Cube}} \approx \frac{\text{No. Points in Sphere}}{\text{No. of Points in Cube}}$$

By symmetry we get the same result if work in a single quadrant, say $[0,1]\times[0,1]\times[0,1]\times[0,1]$.

Here is the calculation in Scilab. First we write a function `monte4d` to do the calculation:

```
function v = monte4d(n)
  k = 0              \\ count of number of points in sphere
  for i = 1:n
    x = rand(1,4)        \\ x = point in unit cube
    if (norm(x) <= 1)    \\ point x lies in sphere
      k = k+1
    end
  end
  v = 16*k/n             \\ 16 quadrants!
endfunction

-->monte4d(100000)
 ans  =

    4.91392
```

By the way, the exact answer is $\pi^2/2 = 4.9348$

14.1.4 Multiple Integrals

Generalizing the formula for Monte-Carlo integration in one dimension, we have a formula for Monte-Carlo integration in any number of dimensions: if

$$I = \int_\Omega f\, dV$$

then

$$I \approx \text{Volume of } \Omega \times \text{Average Value of } f \text{ in } \Omega$$

For one-dimensional integration problems Monte-Carlo integration is quite inefficient. For high-dimensional integration it is a useful technique. The reasons for this are twofold:

1. The fact that error is proportional to $1/\sqrt{n}$ does not depend on the dimension. In other words it performs just as well in high dimensions as in one dimension.

2. It can easily handle regions with irregular boundaries. The method used in the previous section to estimate volumes can easily be adapted to Monte-Carlo integration over any region.

Example

We will evaluate the integral

$$I = \iint_\Omega \sin \sqrt{\ln(x+y+1)}\, dx\, dy$$

where Ω is the disk

$$\left(x - \frac{1}{2}\right)^2 + \left(y - \frac{1}{2}\right)^2 \leq \frac{1}{4}$$

Since the disk Ω is contained within the square $[0,1] \times [0,1]$, we can generate x and y as uniform $[0,1]$ random numbers, and keep those which lie in the disk Ω.

```
function ii = monte2da(n)
  k = 0              // count no. of points in disk
  sumf = 0           // keep running sum of function values
  while (k < n)      // keep going until we get n points
    x = rand(1,1)
    y = rand(1,1)
    if ((x-0.5)^2 + (y-0.5)^2 <= 0.25) then  // (x,y) is in disk
      k = k + 1                              // increment count
      sumf = sumf + sin(sqrt(log(x+y+1)))    // increment sumf
    end
  end
  ii = (%pi/4)*(sumf/n)     // %pi/4 = volume of disk
endfunction

-->monte2da(100000)
 ans  =

    0.5679196
```

Example

In the Monte-Carlo approximation

$$I \approx \text{Volume of } \Omega \times \text{Average Value of } f \text{ in } \Omega$$

we can estimate the volume of the region Ω at the same time as we estimate average the function f.

We generate points in a volume V – usually rectangular – containing Ω. If we generate n points in V of which k lie in the region Ω then

$$\text{Volume } \Omega \approx \frac{k}{n} \text{ Volume } V$$

Since

$$\text{Average Value of } f \text{ in } \Omega \approx \frac{1}{k} \sum f$$

the k's cancel and we have

$$I \approx \text{Volume } V \times \frac{1}{n} \sum f.$$

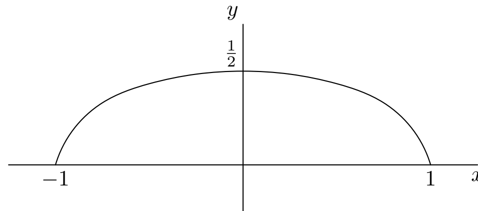
where the sum is over the points lying in the region Ω.

We will now evaluate the integral

$$I = \iint_{\Omega} y \, dx dy$$

over the semi-elliptical region Ω given by

$$x^2 + 4y^2 \leq 1, \quad y \geq 0$$



Although there is a simple formula for the area of an ellipse, we will use Monte-Carlo to estimate the area as we perform the integration. Since the region Ω is contained within the rectangle $[-1, 1] \times [0, 1/2]$, we generate x and y as uniform $[-1, 1]$ and uniform $[0, 1/2]$ random numbers respectively. We will generate `n` random numbers, and use those lying in Ω to form the sum of the function. Note that the rectangular region has area $= 1$.

```
function ii = monte2db(n)
  sumf = 0            // keep running sum of function values
  for i = 1:n
    x = 2*rand(1,1) - 1       // x in [-1,1]
    y = rand(1,1)/2           // y in [0,1/2]
    if (x^2 + 4*y^2 <= 1)     // point lies in region
      sumf = sumf + y         // increment sumf
    end
```

```
  end
  ii = sumf/n                       // Volume = 1
endfunction

-->monte2db(100000)
 ans  =

    0.1668539
```

The exact value is 1/6.

14.2 Simulation

14.2.1 Two Dice

Simulation using random numbers is a technique for estimating probabilities. It is best understood by looking at examples.

The first example is an easy one which will illustrate the effective use of Scilab. Suppose I toss two unbiased dice. What is the probability that the sum of the numbers showing is less than or equal to 4? A simple counting argument shows that the answer is $6/36 = .16667$.

We will simulate 10 tosses of the dice. The number showing on each die is an integer uniformly distributed between 1 and 6. We saw how generate such random numbers in Lecture 13, and we will represent the results of our simulation by a 2×10 array.

```
-->x = floor(6*rand(2,10) + 1)
 x  =

!   2.    4.    1.    6.    2.    3.    6.    3.    5.    5. !
!   2.    1.    1.    1.    5.    6.    6.    5.    2.    6. !
```

Now we can calculate the sum of the two dice, noting that `sum(x,'r')` sums over the *columns* of the matrix `x` while `sum(x,'c')` sums over the *rows*:

```
-->s = sum(x,'r')
 s  =

!   4.    5.    2.    7.    7.    9.    12.    8.    7.    11. !
```

What we need now is to count how many of these are ≤ 4. We can use **boolean arrays** to do this efficiently:

```
-->ss = (s <= 4)
 ss  =

! T F T F F F F F F F !
```

Here T and F stand for true and false, and this array just tells us whether the corresponding elements of s are ≤ 4. The boolean elements T and F also have the numerical values 1 and 0 respectively, so we can find the number of cases in which the sum of the two dice is ≤ 4 by taking the sum of this array.

```
-->xx = sum(ss)
 xx  =

    2.
```

So in 2 of our 10 simulations, the sum was less than or equal to four.

We can write a function to do this:

```
function y = dice2(n)
  x = floor(6*rand(2,n) + 1)
  s = sum(x, 'r')
  ss = (s <= 4)
  y = sum(ss)/n
endfunction

-->dice2(100000)
 ans  =

    0.16727
```

14.2.2 Errors

The discussion of errors and the central limit theorem also applicable to general simulation problems. The expected error in n trials is

$$\text{Error} = \frac{\sigma}{\sqrt{n}}$$

Unfortunately we do not usually know the value of σ which is the variance of the probability we are estimating. It is usually possible to get an estimate of σ from the simulation itself, but we will mainly be concerned with the $n^{-1/2}$ factor which determines how the error depends on n.

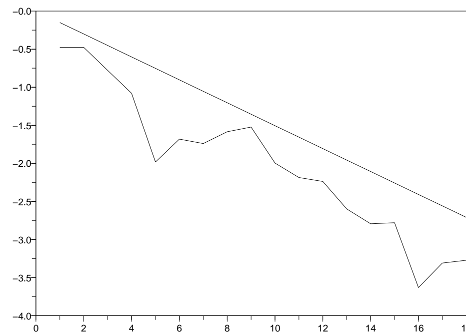
Let us look at the error in our dice problem for $2, 4, \ldots, 2^{18}$ trials:

```
-->dn = zeros(1,18);

-->for k = 1:18
-->  dn(k) = dice2(2^k);
-->end

-->dn
```

```
 dn  =


         column 1 to 7

!   0.5    0.5    0.    0.25    0.15625    0.1875    0.1484375 !

         column  8 to 11

!   0.140625    0.1367188    0.1767578    0.1601562 !

         column 12 to 15

!   0.1608887    0.1691895    0.1682739    0.1650085 !

         column 16 to 18

!   0.1669006    0.1661758    0.1661339 !
```

According to the theory the error should be proportional to $n^{-1/2}$. Simply superimposing a line of slope -1/2 should indicate whether the error really is decreasing as theorized.

```
-->err = abs(dn-1/6);

-->n = 2 .^(1:18);

-->ee = n.^{-1/2};

-->plot2d(log10(err))

-->plot2d(log10(ee))
```



14.2.3 Two Loaded Dice

Suppose now that our dice are loaded, so that the probabilities are:

Outcome	1	2	3	4	5	6
Probability	.20	.14	.22	.16	.17	.11
Cumulative	.20	.34	.56	.72	.89	1.00

How can we simulate tossing these dice? Suppose x is uniformly distributed on $[0, 1]$. If $x \in [0, .20]$ we assign the value 1, if $x \in [.20, .34]$ we assign the value 2 etc. A Scilab function to assign these probabilities is easy to write using `if-then-else` statements, but we can do better by using boolean arrays:

```
function y = ldice(x)
  x1 = (x >= 0.20)
  x2 = (x >= 0.34)
  x3 = (x >= 0.56)
  x4 = (x >= 0.72)
  x5 = (x >= 0.89)
  y = 1*x1 + 1*x2 + 1*x3 + 1*x4 + 1*x5  + 1
endfunction
```

In the function above, each of `x1` to `x5` is a boolean vector, we multiply `1` to convert to a numerical value and the add the results.

```
-->x = rand(1,8)
 x  =


         column 1 to 4

!   0.0976644    0.8918166    0.1762568    0.0862309 !

         column 5 to 8

!   0.2136821    0.9119316    0.5557921    0.5549552 !

-->ldice(x)
 ans  =

!   1.    6.    1.    1.    2.    6.    3.    3. !
```

Our function to estimate the probability that the sum of the two dice is less than or equal to 4 is nearly the same as the previous example:

```
function y = ldice2(n)
  x = ldice(rand(2,n))
  s = sum(x, 'r')
  ss = (s <= 4)
  y = sum(ss)/n
endfunction

-->ldice2(100000)
 ans  =

    0.20316
```

You might like to calculate the exact answer and compare.

14.2.4 The Birthday Problem

Suppose we have N people in room. What is the probability that (at least) two people share the same birthday.

To solve this by simulation we can proceed as follows:

1. Generate N random birthdays.
2. Check if two coincide.

Generating the random birthdays is easy, just generate a random vector of integers in the range 1 to 365 (we will ignore leap years). To check whether such a vector contains two numbers the same, we first sort the birthdays, and then only have to check whether neighbouring components of the vector are equal.

```
function p = birthdays(n, trials)
  k = 0
  for i = 1:trials
    bs = floor(365 * rand(1,n) + 1)
    bs = sort(bs)
    for j = 1:(n-1)
      if (bs(j) == bs(j+1))    // we have found a match
        k = k+1                // increment count
        break                  // break out of for loop
      end
    end
  end
  p = k/trials
endfunction
```

Only 23 people are needed for the probability that two have the same birthday to be greater than 0.5. Let us check this:

```
-->birthdays(22, 100000)
 ans  =

    0.47559

-->birthdays(23, 100000)
 ans  =

    0.50831

-->birthdays(24, 100000)
 ans  =

    0.53805
```

For $N = 100$ it is almost certain that two people will share a birthday:

```
-->birthdays(100, 100000)
 ans  =

    1.
```

14.3 Appendix – Boolean Matrices

The boolean values are `%t`, printed `T`, for **true** and `%f`, printed `F`, for **false**. Matrices, and particularly vectors, of boolean values are often useful in simulation.

14.3.1 Comparison and Logical Operators

Boolean matrices are usually constructed by applying comparison operators:

`==`	equal
`~=`	not equal
`<`	less than
`>`	greater than
`<=`	less than or equal to
`>=`	greater than or equal to

to real matrices.

```
-->a=rand(4,4)
 a  =

!   0.2806498    0.1121355    0.8415518    0.1138360 !
!   0.1280058    0.6856896    0.4062025    0.1998338 !
```

```
!   0.7783129    0.1531217    0.4094825    0.5618661 !
!   0.2119030    0.6970851    0.8784126    0.5896177 !

-->bb = a > 0.4
 bb  =

! F F T F !
! F T T F !
! T F T T !
! F T T T !
```

The usual matrix operations, `+`, `-` and `*` are undefined for boolean matrices. The logical operators

`&`	and
`|`	or
`~`	not

apply element by element to boolean matrices.

```
-->~bb
 ans  =

! T T F T !
! T F F T !
! F T F F !
! T F F F !
```

14.3.2 Real and Boolean Matrices

Boolean matrices may be converted to real 0-1 matrices, 0 = false, 1 = true, with the `bool2s` command:

```
-->bool2s(bb)
 ans  =

!   0.    0.    1.    0. !
!   0.    1.    1.    0. !
!   1.    0.    1.    1. !
!   0.    1.    1.    1. !

-->sum(ans)
 ans  =

    9.
```

We could have got the same result with `sum(bb)`, though this would seem to contradict the rule that we can't add booleans. However this can be quite handy for counting the number of components of a vector or matrix satisfying some condition, e.g.

```
-->sum(a > 0.4)
 ans  =

    9.
```

Scilab allows arithmetic operations where one operand is boolean and the other a number. In this case the boolean values are converted to 0-1 values. For example;

```
-->bb
 bb  =

! F F T F !
! F T T F !
! T F T T !
! F T T T !

-->bb*4
 ans  =

!   0.    0.    4.    0. !
!   0.    4.    4.    0. !
!   4.    0.    4.    4. !
!   0.    4.    4.    4. !

-->bb-1
 ans  =

! - 1.  - 1.    0.  - 1. !
! - 1.    0.    0.  - 1. !
!   0.  - 1.    0.    0. !
! - 1.    0.    0.    0. !
```

14.3.3 `find`

The `find` operator returns the indices of the true components of boolean vector or matrix. It is most useful for vectors, so we will concentrate on those:

```
-->a = rand(1,12)
 a  =


         column 1 to 4

!   0.5878720    0.4829179    0.2232865    0.8400886 !

         column 5 to 8

!   0.1205996    0.2855364    0.8607515    0.8494102 !

         column  9 to 12

!   0.5257061    0.9931210    0.6488563    0.9923191 !

-->bb = a > 0.4
 bb  =

! T T F T F F T T T T T T !

-->find(bb)
 ans  =

!   1.    2.    4.    7.    8.    9.    10.    11.    12. !
```

We can omit the intermediate step of constructing the boolean vector:

```
-->ii = find(a > 0.4)
 ii  =

!   1.    2.    4.    7.    8.    9.    10.    11.    12. !
```

The vector of indices constructed in this way can be used to extract the corresponding components of the vector, in this case all the components greater than 0.4:

```
-->a(ii)
 ans  =


         column 1 to 4

!   0.5878720    0.4829179    0.8400886    0.8607515 !
```

```
         column 5 to 8

!   0.8494102    0.5257061    0.9931210    0.6488563 !

         column 9

!   0.9923191 !
```