Programming and Numerical Analysis

integral method of function

筒井 広明 / TSUTSUI Hiroaki

4.0 Introduction

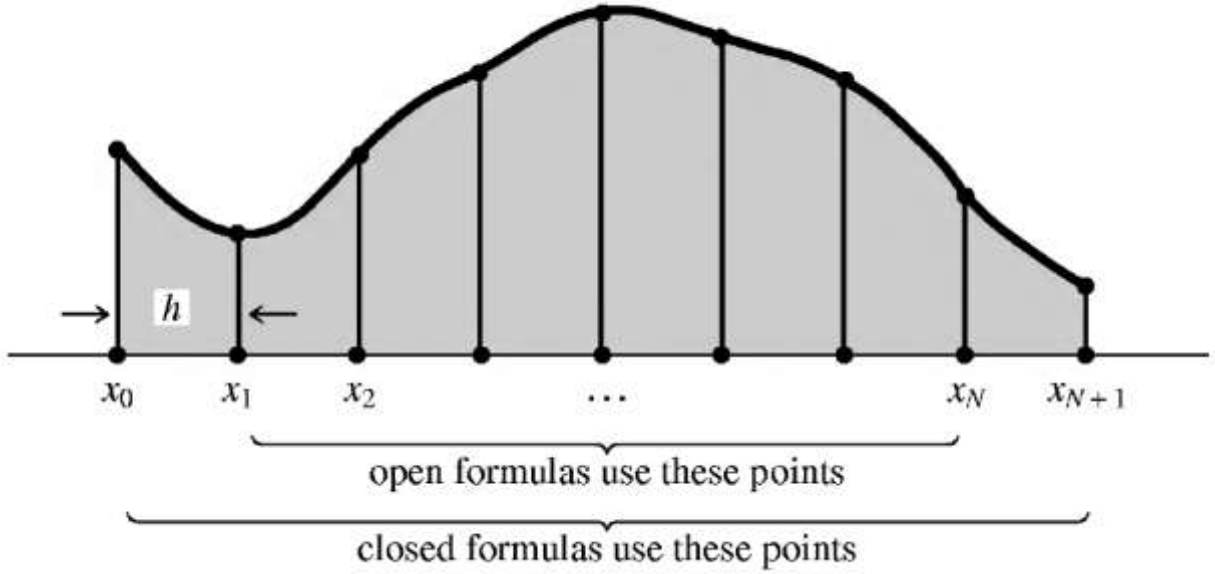
Numerical integration, which is also called *quadrature*, has a history extending back to the invention of calculus and before. With the invention of automatic computing, quadrature became just one numerical task among many, and not a very interesting one at that.

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

The quadrature methods in this chapter are based, in one way or another, on the obvious device of adding up the value of the integrand at a sequence of abscissas within the range of integration. The game is to obtain the integral as accurately as possible with the smallest number of function evaluations of the integrand. Just as in the case of interpolation (Chapter 3), one has the freedom to choose methods of various *orders*, with higher order sometimes, but not always, giving higher Accuracy.

4.1 Classical Formulas for *Equally Spaced* Abscissas

- Where would any book on numerical analysis be without Mr. Simpson and his “rule”? The classical formulas for integrating a function whose value is known at *equally spaced* steps have a certain elegance about them, and they are redolent with historical association. Through them, the modern numerical analyst communes with the spirits of his or her predecessors back across the centuries, as far as the time of Newton, if not farther.



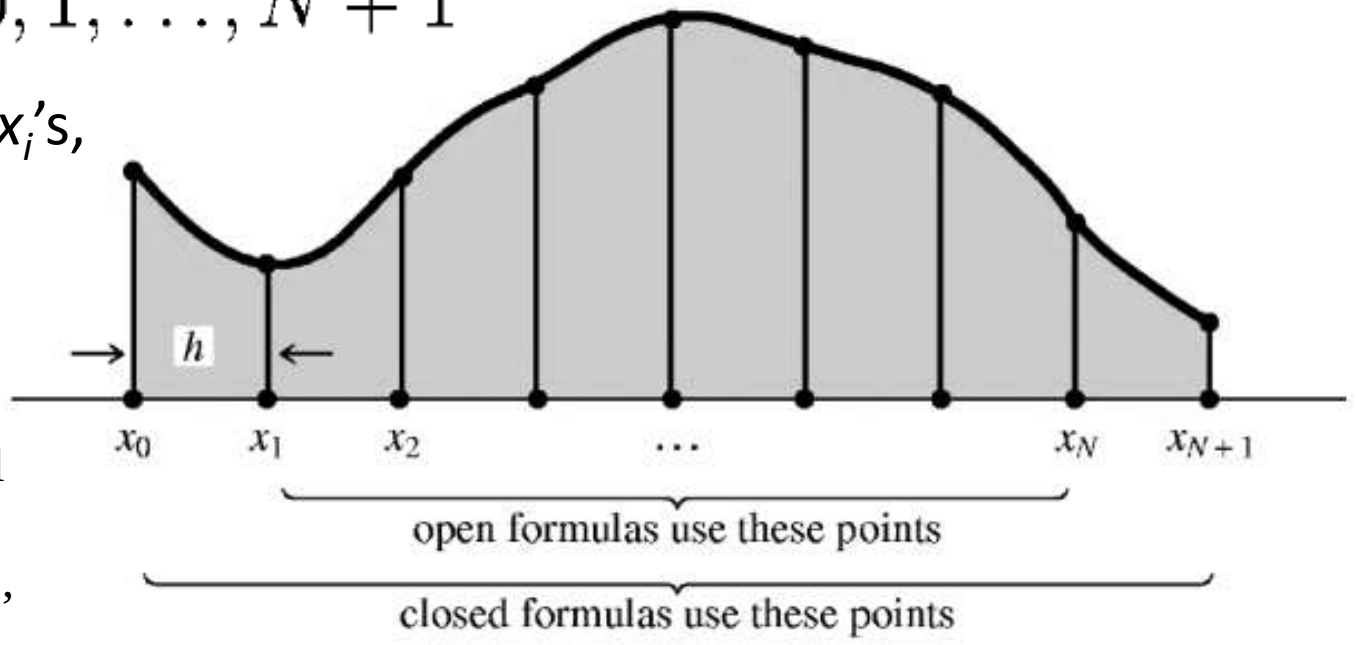
Classical Formulas for Equally Spaced Abscissas

Some notation: We have a sequence of abscissas, denoted $x_0, x_1, \ldots, x_N$, x_{N+1} which are spaced apart by a constant step h,

$$x_i = x_0 + ih \qquad i = 0, 1, \ldots, N+1$$

A function $f(x)$ has known values at the x_i's,

$$f(x_i) \equiv f_i$$



We want to integrate the function $f(x)$ between a lower limit a and an upper limit b, where a and b are each equal to one or the other of the x_i's. An integration formula that uses the value of the function at the endpoints, $f(a)$ or $f(b)$, is called a *closed* formula.

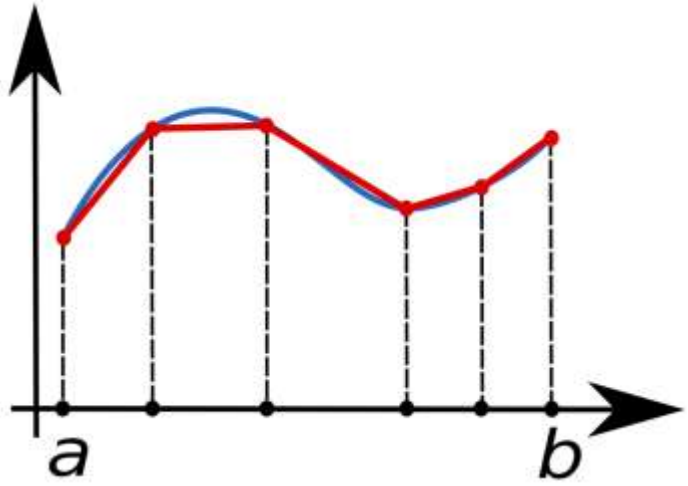
The basic building blocks of the classical formulas are rules for integrating a function over a small number of intervals. As that number increases, we can find rules that are exact for polynomials of increasingly high order.

Closed Newton-Cotes Formulas

- *Trapezoidal rule:*

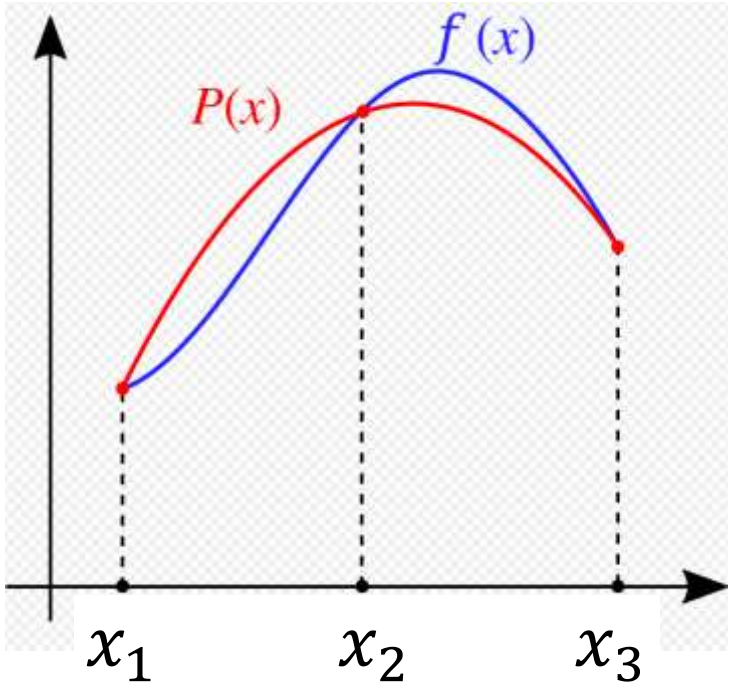
$$\int_{x_1}^{x_2} f(x)dx = h\left[\frac{1}{2}f_1 + \frac{1}{2}f_2\right] \quad + O(h^3 f'') \qquad (4.1.3)$$

Here the error term *O*( ) signifies that the true answer differs from the estimate by an amount that is the product of some numerical coefficient times h^3 times the value of the function's second derivative somewhere in the interval of integration. The coefficient is knowable, and it can be found in all the standard references on this subject.



Simpson's rule:

Equation (4.1.3) is a two-point formula (x_1 and x_2). It is exact for polynomials up to and including degree 1, i.e., $f(x) = x$. One anticipates that there is a three-point formula exact up to polynomials of degree 2. This is true; moreover, by a cancellation of coefficients due to left-right symmetry of the formula, the three-point formula is exact for polynomials up to and including degree 3, i.e., $f(x) = x^3$:



Simpson's rule can be derived by approximating the integrand $f(x)$ by the quadratic interpolant $P(x)$.

$$\int_{x_1}^{x_3} f(x)dx = h\left[\frac{1}{3}f_1 + \frac{4}{3}f_2 + \frac{1}{3}f_3\right] + O(h^5 f^{(4)})$$

$$h = x_3 - x_2 = x_2 - x_1$$

Here $f^{(4)}$ means the fourth derivative of the function f evaluated at an unknown place in the interval. Note also that the formula gives the integral over an interval of size $2h$, so the coefficients add up to 2.

$$\frac{1}{3} + \frac{4}{3} + \frac{1}{3} = 2$$

Boole's (Bode's) rule:

The five-point formula again benefits from a cancellation:

$$\int_{x_1}^{x_5} f(x)dx = h\left[\frac{14}{45}f_1 + \frac{64}{45}f_2 + \frac{24}{45}f_3 + \frac{64}{45}f_4 + \frac{14}{45}f_5\right] \quad + O(h^7 f^{(6)})$$

This is exact for polynomials up to and including degree 5.
At this point the formulas stop being named after famous personages, so we will not go any further.

4.5 Gaussian Quadratures and Orthogonal Polynomials

- The idea of *Gaussian quadratures* is to give ourselves the freedom to choose not only the weighting coefficients, but also the **location of the abscissas** at which the function is to be evaluated: They will **no longer be equally spaced**.

- Thus, we will have *twice* the number of degrees of freedom at our disposal; it will turn out that we can achieve **Gaussian quadrature formulas** whose order is, essentially, twice that of the Newton-Cotes formula with the same number of function evaluations.

Gaussian Quadratures and Orthogonal Polynomials

- There is, however, one additional feature of Gaussian quadrature formulas that adds to their usefulness: We can arrange the choice of weights and abscissas to make the integral exact for a class of integrands "polynomials times some known function $W(x)$" rather than for the usual class of integrands "polynomials."
- The function $W(x)$ can then be chosen to remove integrable singularities from the desired integral.

Given $W(x)$, in other words, and given an integer N, we can find a set of weights w_j and abscissas x_j such that the approximation

$$\int_a^b W(x) f(x) dx \approx \sum_{j=1}^{N} w_j f(x_j) \qquad (4.5.1)$$

is exact if $f(x)$ is a polynomial.

Gaussian Quadratures and Orthogonal Polynomials

- Notice that the integration formula (4.5.1) can also be written with the weight function $W(x)$ not overtly visible: Define $g(x) \equiv W(x)f(x)$ and $v_j \equiv w_j/W(x_j)$.
- Then (4.5.1) becomes

$$\int_a^b g(x)dx \approx \sum_{j=1}^{N} v_j g(x_j) \qquad (4.5.4)$$

Where did the function $W(x)$ go? It is lurking there, ready to give high-order accuracy to integrands of the form polynomials times $W(x)$, and ready to *deny* high order accuracy to integrands that are otherwise perfectly smooth and well-behaved. When you find tabulations of the weights and abscissas for a given $W(x)$, you have to determine carefully whether they are to be used with a formula in the form of (4.5.1), or like (4.5.4).

Gauss-Legendre

- Here is an example of a quadrature routine that contains the tabulated abscissas and weights for the case $W(x) = 1$ and $N = 10$.

```
float qgaus(float (*func)(float), float a, float b)
Returns the integral of the function func between a and b, by ten-point Gauss-Legendre inte-
gration: the function is evaluated exactly ten times at interior points in the range of integration.
{
    int j;
    float xr,xm,dx,s;
    static float x[]={0.0,0.1488743389,0.4333953941,      The abscissas and weights.
        0.6794095682,0.8650633666,0.9739065285};            First value of each array
    static float w[]={0.0,0.2955242247,0.2692667193,        not used.
        0.2190863625,0.1494513491,0.0666713443};

    xm=0.5*(b+a);
    xr=0.5*(b-a);
    s=0;                              Will be twice the average value of the function, since the
    for (j=1;j<=5;j++) {                  ten weights (five numbers above each used twice)
        dx=xr*x[j];                       sum to 2.
        s += w[j]*((*func)(xm+dx)+(*func)(xm-dx));
    }
    return s *= xr;                   Scale the answer to the range of integration.
}
```

The above routine illustrates that one can use Gaussian quadratures without necessarily understanding the theory behind them: One just locates tabulated weights and abscissas in a book.

Gauss-Legendre

$$W(x) = 1 \qquad -1 < x < 1$$

$$(j+1)P_{j+1} = (2j+1)xP_j - jP_{j-1}$$

$$w_j = \frac{2}{(1-x_j^2)[P'_N(x_j)]^2}$$

$$\int_{x_1}^{x_2} f(x)dx = \sum_{j=1}^{N} w_j f(x_j)$$

```
#include <math.h>
#define EPS 3.0e-11                          EPS is the relative precision.

void gauleg(float x1, float x2, float x[], float w[], int n)
Given the lower and upper limits of integration x1 and x2, and given n, this routine returns
arrays x[1..n] and w[1..n] of length n, containing the abscissas and weights of the Gauss-
Legendre n-point quadrature formula.
{
    int m,j,i;
    double z1,z,xm,xl,pp,p3,p2,p1;           High precision is a good idea for this rou-
                                                 tine.
    m=(n+1)/2;                               The roots are symmetric in the interval, so
    xm=0.5*(x2+x1);                              we only have to find half of them.
    xl=0.5*(x2-x1);
    for (i=1;i<=m;i++) {                     Loop over the desired roots.
        z=cos(3.141592654*(i-0.25)/(n+0.5));
        Starting with the above approximation to the ith root, we enter the main loop of
        refinement by Newton's method.
        do {
            p1=1.0;
            p2=0.0;
            for (j=1;j<=n;j++) {             Loop up the recurrence relation to get the
                p3=p2;                           Legendre polynomial evaluated at z.
                p2=p1;
                p1=((2.0*j-1.0)*z*p2-(j-1.0)*p3)/j;
            }
            p1 is now the desired Legendre polynomial. We next compute pp, its derivative,
            by a standard relation involving also p2, the polynomial of one lower order.
            pp=n*(z*p1-p2)/(z*z-1.0);
            z1=z;
            z=z1-p1/pp;                      Newton's method.
        } while (fabs(z-z1) > EPS);
        x[i]=xm-xl*z;                        Scale the root to the desired interval,
        x[n+1-i]=xm+xl*z;                    and put in its symmetric counterpart.
        w[i]=2.0*xl/((1.0-z*z)*pp*pp);       Compute the weight
        w[n+1-i]=w[i];                       and its symmetric counterpart.
    }
}
```

Programming and Numerical Analysis

random number, Monte Carlo method

筒井 広明 / TSUTSUI Hiroaki

7.0 Introduction

- It may seem perverse to use a computer, that most precise and deterministic of all machines conceived by the human mind, to produce “random” numbers.
- One sometimes hears computer-generated sequences termed *pseudo-random*, while the word *random* is reserved for the output of an intrinsically random physical process.

7.1 Uniform Deviates

- **Uniform deviates** are just random numbers that lie within a specified range (typically 0 to 1), with any one number in the range just as likely as any other. They are, in other words, what you probably think "**random numbers**" are.
- However, we want to distinguish uniform deviates from other sorts of random numbers, for example numbers drawn from a normal (Gaussian) distribution of specified mean and standard deviation.

- These other sorts of deviates are almost always generated by performing appropriate operations on one or more uniform deviates.

System-Supplied Random Number Generators

- Most C implementations have, lurking within, a pair of library routines for initializing, and then generating, "random numbers." In ANSI C, the synopsis is:

```
#include <stdlib.h>
#define RAND_MAX ...
void srand(unsigned seed);
int rand(void);
```

If you want a random float value between 0.0 (inclusive) and 1.0 (exclusive), you get it by an expression like

```
x = rand()/(RAND_MAX+1.0);
```

You initialize the random number generator by invoking srand(seed) with some arbitrary seed. Each initializing value will typically result in a different random sequence, or a least a different starting point in some one enormously long sequence. The *same* initializing value of seed will always return the *same* random sequence, however.

Now our first, and perhaps most important, lesson in this chapter is: be very, very suspicious of a system-supplied rand() that resembles the one just described.

linear congruential generators

System-supplied rand()s are almost always ***linear congruential generators***, which generate a sequence of integers $I_1, I_2, I_3, \ldots$, each between 0 and $m - 1$ (e.g., RAND_MAX) by the recurrence relation

$$I_{j+1} = aI_j + c \pmod{m} \qquad (7.1.1)$$

3 5 13

$aI_j + c$ $\qquad$ I_{j+1}

$(3 \times 8 + 5) \bmod 13 = 3$

$(3 \times 3 + 5) \bmod 13 = 1$

$(3 \times 1 + 5) \bmod 13 = 8$

Here *m* is called the *modulus*, and *a* and *c* are positive integers called the *multiplier* and the *increment* respectively.

All possible integers between 0 and *m*− 1 occur at some point, so any initial "seed" choice of I_0 is as good as any other: the sequence just takes off from that point.

Portable Random Number Generators

- There is good evidence, both theoretical and empirical, that the simple **multiplicative congruential algorithm**

$$I_{j+1} = aI_j \pmod{m} \qquad (7.1.2)$$

can be as good as any of the more general linear congruential generators that have $c = 0$ (equation 7.1.1) — *if* the multiplier a and modulus m are chosen exquisitely carefully. Park and Miller propose a "**Minimal Standard**" generator based on the choices

$$a = 7^5 = 16807 \qquad m = 2^{31} - 1 = 2147483647 \qquad (7.1.3)$$

It is not possible to implement equations (7.1.2) and (7.1.3) directly in a high-level language, since the product of a and $m - 1$ exceeds the maximum value for a 32-bit integer. A trick due to Schrage for multiplying two 32-bit integers modulo a 32-bit constant, without using any intermediates larger than 32 bits (including a sign bit) is therefore extremely interesting:

Schrage's algorithm

Schrage's algorithm is based on an *approximate factorization* of *m*,

$$m = aq + r, \quad \text{i.e.,} \quad q = [m/a],\ r = m \bmod a$$

with square brackets denoting integer part. If *r* is small, specifically $r < q$, and $0 < z < m - 1$, it can be shown that both $a(z \bmod q)$ and $r[z/q]$ lie in the range $0, \ldots, m - 1$, and that

$$az \bmod m = \begin{cases} a(z \bmod q) - r[z/q] & \text{if it is} \geq 0, \\ a(z \bmod q) - r[z/q] + m & \text{otherwise} \end{cases}$$

The application of Schrage's algorithm to the constants (7.1.3) uses the values $q = 127773$ and $r = 2836$.

$$aq + r = 16807 \times 127773 + 2836 = 2147483647 = 2^{31} - 1 = m$$

ran0() / Schrage's algorithm

```
#define IA 16807
#define IM 2147483647
#define AM (1.0/IM)
#define IQ 127773
#define IR 2836
#define MASK 123459876
float ran0(long *idum)

{
long k;
float ans;
*idum ^= MASK;
k=(*idum)/IQ;
*idum=IA*(*idum-k*IQ)-IR*k;
(*idum < 0) *idum += IM;
ans=AM*(*idum);
*idum ^= MASK;
return ans;
}
```

"Minimal" random number generator of Park and Miller. Returns a uniform random deviate between 0.0 and 1.0. Set or reset idum to any integer value (except the unlikely value MASK) to initialize the sequence; idum must not be altered between calls for successive deviates in a sequence.

XORing with MASK allows use of zero and other simple bit patterns for idum.
Compute idum=(IA*idum) % IM without overif flows by Schrage's method.
Convert idum to a floating result.
Unmask before return.

7.2 Transformation Method

In the previous section, we learned how to generate random deviates with a uniform probability distribution, so that the probability of generating a number between x and $x + dx$, denoted $p(x)dx$, is given by

$$p(x)dx = \begin{cases} dx & 0 < x < 1 \\ 0 & \text{otherwise} \end{cases} \qquad (7.2.1) \qquad \int_{-\infty}^{\infty} p(x)dx = 1$$

Now suppose that we generate a uniform deviate x and then take some prescribed function of it, $y(x)$. The probability distribution of y, denoted $p(y)dy$, is determined by the fundamental transformation law of probabilities, which is simply

$$|p(y)dy| = |p(x)dx| \quad \Rightarrow \quad p(y) = p(x)\left|\frac{dx}{dy}\right|$$

Exponential Deviates

- As an example, suppose that $y(x) \equiv -\ln(x)$, and that $p(x)$ is as given by equation (7.2.1) for a uniform deviate. Then

$$p(y)dy = \left|\frac{dx}{dy}\right| dy = e^{-y}dy$$

$$p(x)dx = \begin{cases} dx & 0 < x < 1 \\ 0 & \text{otherwise} \end{cases}$$

$$p(y) = p(x)\left|\frac{dx}{dy}\right|$$

which is distributed exponentially.

transformation method

$$p(x)dx = \begin{cases} dx & 0 < x < 1 \\ 0 & \text{otherwise} \end{cases}$$

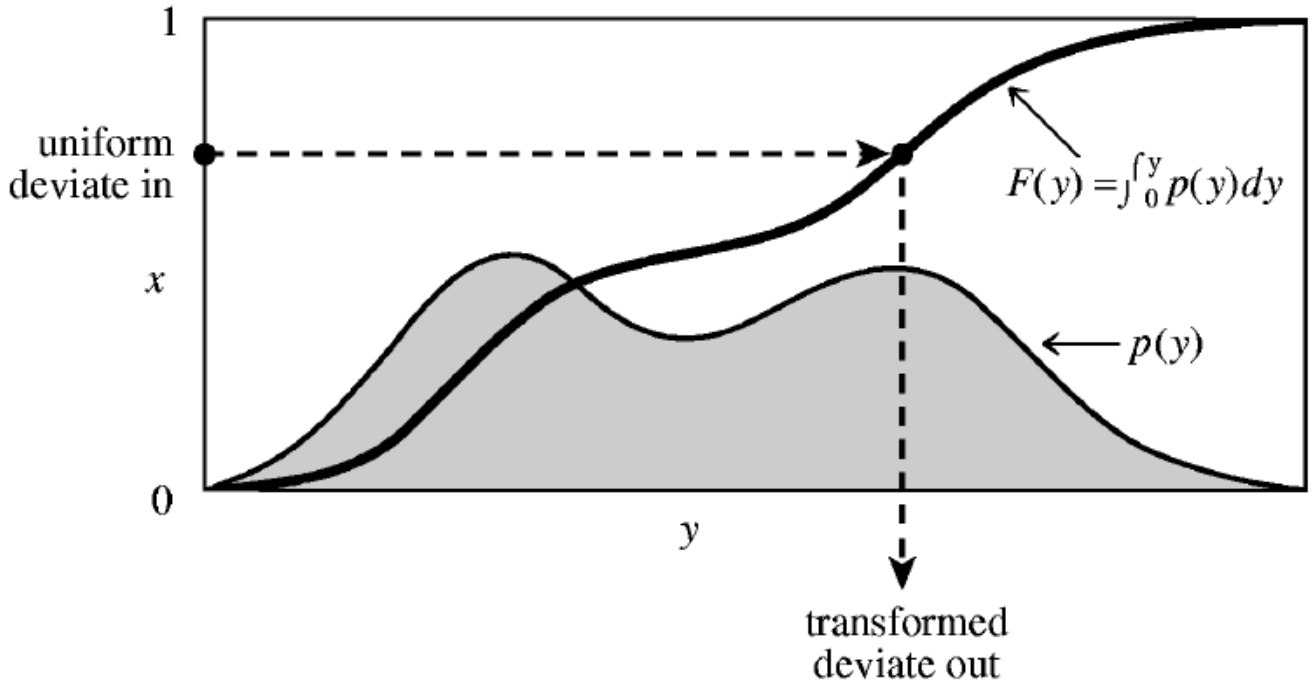
$$p(y) = p(x)\left|\frac{dx}{dy}\right| \qquad (7.2.4)$$

Let's see what is involved in using the above transformation method to generate some arbitrary desired distribution of *y*'s, say one with $p(y) = f(y)$ for some positive function f whose integral is 1. According to (7.2.4), we need to solve the differential equation

$$\frac{dx}{dy} = f(y)$$

But the solution of this is just $x = F(y)$, where $F(y)$ is the indefinite integral of $f(y)$. The desired transformation which takes a uniform deviate into one distributed as $f(y)$ is therefore

$$y(x) = F^{-1}(x)$$



Normal (Gaussian) Deviates

If $x_1, x_2, \ldots$ are random deviates with a *joint* probability distribution $p(x_1, x_2, \ldots)$ $dx_1 dx_2 \ldots$, and if $y_1, y_2, \ldots$ are each functions of all the x's (same number of y's as x's), then the joint probability distribution of the y's is

$$p(y_1, y_2, \ldots) dy_1 dy_2 \ldots = p(x_1, x_2, \ldots) \left| \frac{\partial(x_1, x_2, \ldots)}{\partial(y_1, y_2, \ldots)} \right| dy_1 dy_2 \ldots \qquad (7.2.8)$$

where $|\partial(\,)/\partial(\,)|$ is the Jacobian determinant.

An important example of the use of (7.2.8) is the ***Box-Muller* method** for generating random deviates with a normal (Gaussian) distribution,

$$p(y)dy = \frac{1}{\sqrt{2\pi}} e^{-y^2/2} dy$$

Box-Muller method

Consider the transformation between two uniform deviates on (0,1), x_1, x_2, and two quantities y_1, y_2,

$$y_1 = \sqrt{-2\ln x_1}\cos 2\pi x_2$$

$$y_2 = \sqrt{-2\ln x_1}\sin 2\pi x_2$$

$$x_1 = \exp\left[-\frac{1}{2}(y_1^2 + y_2^2)\right]$$

$$x_2 = \frac{1}{2\pi}\arctan\frac{y_2}{y_1}$$

Now the Jacobian determinant can readily be calculated (try it!):

$$\frac{\partial(x_1, x_2)}{\partial(y_1, y_2)} = \begin{vmatrix} \frac{\partial x_1}{\partial y_1} & \frac{\partial x_1}{\partial y_2} \\ \frac{\partial x_2}{\partial y_1} & \frac{\partial x_2}{\partial y_2} \end{vmatrix} = -\left[\frac{1}{\sqrt{2\pi}}e^{-y_1^2/2}\right]\left[\frac{1}{\sqrt{2\pi}}e^{-y_2^2/2}\right]$$

Box-Muller method

$$P(x,y) = \frac{1}{2\pi} e^{-\frac{1}{2}(x^2+y^2)}$$

$$x = r\cos\theta,\ y = r\sin\theta, \qquad \lambda \equiv \frac{r^2}{2}$$

$$P(x,y)dxdy = \frac{d\theta}{2\pi} e^{-\lambda} d\lambda = \frac{d\theta}{2\pi} d\left(-e^{-\lambda}\right) = \frac{d\theta}{2\pi} dz$$

$$z = -e^{-\lambda}$$

$$r^2 = 2\lambda = -2\log|-z|$$

$$x = \sqrt{-2\log u_1}\cos(2\pi u_2)$$

$$y = \sqrt{-2\log u_1}\sin(2\pi u_2)$$

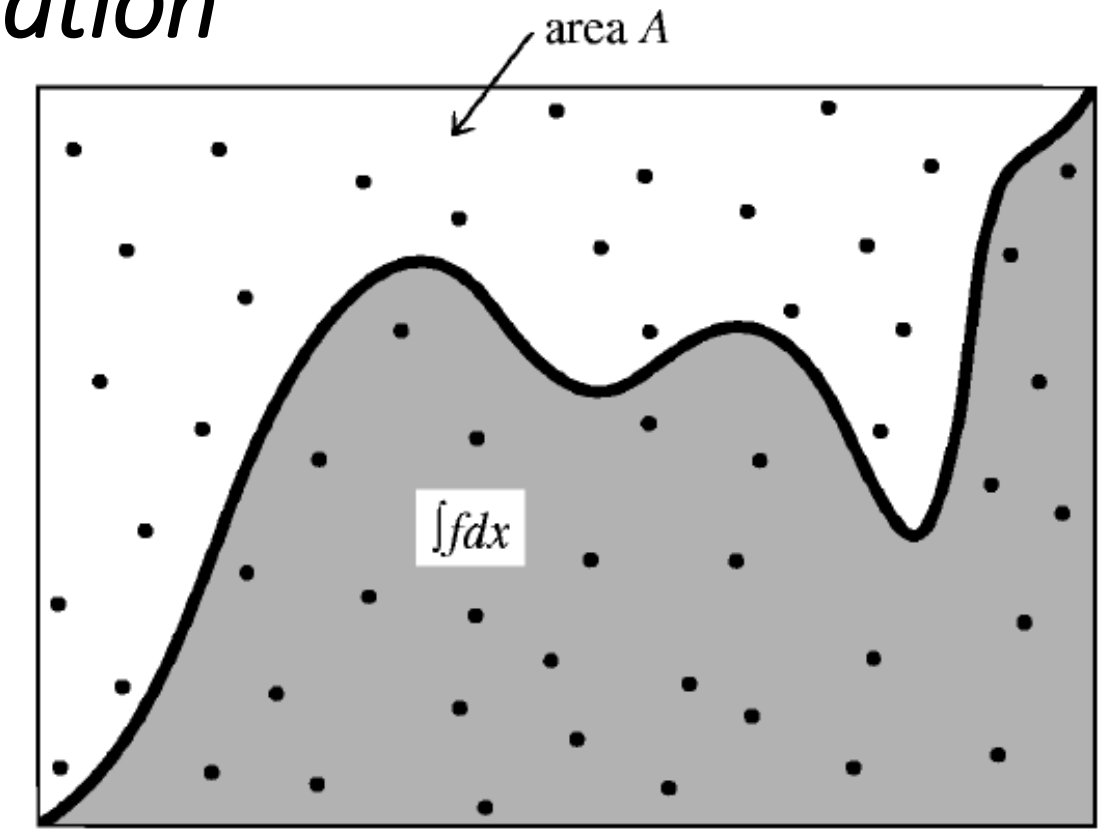
u: uniform random number

7.6 Simple Monte Carlo Integration

Suppose that we pick N random points, uniformly distributed in a multidimensional volume V. Call them $x_1, \ldots, x_N$. Then the basic theorem of Monte Carlo integration estimates the integral of a function f over the multidimensional volume,

$$\int f\, dV \approx V\langle f\rangle \pm V\sqrt{\frac{\langle f^2\rangle - \langle f\rangle^2}{N}} \qquad (7.6.1)$$

$$\langle f\rangle \equiv \frac{1}{N}\sum_{i=1}^{N} f(x_i) \qquad \langle f^2\rangle \equiv \frac{1}{N}\sum_{i=1}^{N} f^2(x_i)$$



Suppose that you want to integrate a function g over a region W that is not easy to sample randomly. For example, W might have a very complicated shape. No problem. Just find a region V that *includes* W and that *can* easily be sampled (Figure), and then define f to be equal to g for points in W and equal to zero for points outside of W (but still inside the sampled V). You want to try to make V enclose W as closely as possible, because the zero values of f will increase the error estimate term of (7.6.1). And well they should: points chosen outside of W have no information content, so the effective value of N, the number of points, is reduced. The error estimate in (7.6.1) takes this into account.

What is Monte Carlo method?

- A method to evaluate stochastic phenomena from random numbers created by computers.

<example> π is obtained.

Computers can create uniform random numbers [0, 1)

$(x_1, y_1), (x_2, y_2), \ldots$