Queens College, CUNY, Department of Computer Science

Numerical Methods

CSCI 361 / 761

Fall 2017

Instructor: Dr. Sateesh Mane

October 7, 2017

7 Lecture 7

This lecture is for your information only.
It contains advanced topics.
The material in this lecture is not for examination.

7.1 Numerical evaluation of integrals (advanced topics)

- Recall that we wish to numerically compute the value of the integral

$$I = \int_a^b f(x)\, dx\,. \tag{7.1.1}$$

- Various algorithms have been described previously.
- This lecture describes advanced topics.
- The material in this lecture is not for examination.

7.2 Quadrature

- Suppose the points in a numerical integration formula are ***not*** evenly spaced.
  This leads us to **quadrature rules.**

- A **quadrature rule** is a formula to approximate the value of the integral of a function $f(x)$ in a domain $a \le x \le b$ by using a weighted sum of the function values at various points in the domain.

- Previously we divided the interval $[a, b]$ into n subintervals, each of length $h = (b - a)/n$. Now let us consider only one subinterval, and analyze algorithms to evaluate the integral of $f(x)$ in one subinterval only. Then we can sum the results for all the subintervals.

- Hence in the formulas below, we might as well say "interval" instead of subinterval. For the sake of formal theory, it is convenient to say that the interval extends from -1 to 1. This is not really a problem. It is easy to transform from $[-1, 1]$ to $[x_i, x_{i+1}]$. Define a linear function

$$g(x) = -x_i \frac{x-1}{2} + x_{i+1} \frac{x+1}{2} \,. \tag{7.2.1}$$

  Then *(work it out for yourself!)*

$$g(-1) = -x_i \frac{(-1-1)}{2} + 0 = x_i \,, \tag{7.2.2a}$$

$$g(1) = 0 + x_{i+1} \frac{1+1}{2} = x_{i+1} \,. \tag{7.2.2b}$$

  So we can transform any interval from $[-1, 1]$ to $[x_i, x_{i+1}]$. The reverse is also easy.

- Suppose we have m points x_i in the interval $[-1, 1]$ and weights w_i $(i = 1, 2, \ldots, m)$. Then we say the value of the integral is approximately

$$\int_a^b f(x)\, dx \simeq \sum_{i=}^{m} w_i f(x_i) \,. \tag{7.2.3}$$

  Stated this way, the midpoint rule, trapezoid rule and Simpson's rule are all quadrature rules.

- **Note that most (all?) textbooks use the notation "n" points instead of m.**

- I used "m" to avoid confusion with the number of subintervals n, which was used in the formulas for the midpoint rule, trapezoid rule and Simpson's rule.

- **Hence be advised that my notation "m" is not standard. Everyone else uses n.**

7.3 Gaussian quadrature

- Gauss is Karl Friedrich Gauss, the famous mathematician.

- We saw previously that using Simpson's rule, we evaluated the function at three points in the domain and it gave the exact result for all polynomials up to degree 3 (cubic polynomials).

- *Can we do better?* Yes we can.
- A **Gaussian quadrature rule** is a formula which evaluates the function at m points in the domain of integration and yields the exact result **for all polynomials up to degree $2m-1$.**
- Hence with 3 points, we can obtain the exact result for all polynomials up to degree **five**.
- **Note:** In the case of the trapezoid rule and Simspon's rule, two of the points are at the end points of the interval (-1 and 1). In the case of the midpoint rule, there is only one point, at the midpoint of the interval (i.e. at $x = 0$ in this case). In general the points can be anywhere, and may not necessarily include the end points of the interval
- ***Why polynomials?***
- Suppose we are given a function $f(x)$. Suppose also that we notice that $f(x)$ can be expressed as a product of a **weight function** $w(x)$ and another function $g(x)$

$$f(x) = w(x)g(x)\,. \tag{7.3.1}$$

  The weight function $w(x)$ is a known function (we choose it). Suppose further that $g(x)$ is approximately a polynomial. Then we can obtain a good approximation for the integral of $f(x)$ by evaluating the function at only a few points. The locations of the points x_i depend on the weight function $w(x)$.
- In other words, given $f(x)$, ***we search around for a suitable weight function*** $w(x)$ ***such that*** $f(x)$ ***can be written as*** $f(x) = w(x)g(x)$***, and*** $g(x)$ ***is approximately a polynomial.*** If we can find a suitable weight function $w(x)$, then we can devise a good Gaussian quadrature rule.
- Common choices for weight functions are

$$w(x) = 1\,, \qquad w(x) = \frac{1}{\sqrt{1-x^2}}\,, \qquad w(x) = e^{-x^2}\,. \tag{7.3.2}$$

- The general theory requires a lot of mathematics. We shall study only the simplest case, which is $w(x) = 1$. In that case, $f(x) = g(x)$, so $f(x)$ by itself must be approximately a polynomial.

7.4 Gauss-Legendre quadrature

- There are many examples of Gaussian quadrature.
- We shall study only the simplest, which is called **Gauss-Legendre quadrature**.
- The weight function is $w(x) = 1$.

7.4.1 One point

- Let us begin with $m = 1$, only one point. Obviously the point x_1 is located at the midpoint of the interval, so $x_1 = 0$. This is the midpoint rule. The weight is $w_1 = 2$, so

$$\int_{-1}^{1} f(x)\,dx \simeq 2f(0)\,. \tag{7.4.1.1}$$

- This should give the exact result for all polynomials up to degree 1 (linear).
- Let us use this to integrate some polynomials, and compare with the exact result.

$$\int_{-1}^{1} 1\,dx = 2\,, \qquad 2f(0) = 2\,, \tag{7.4.1.2a}$$

$$\int_{-1}^{1} x\,dx = 0\,, \qquad 2f(0) = 0\,, \tag{7.4.1.2b}$$

$$\int_{-1}^{1} x^2\,dx = \frac{2}{3}\,, \qquad 2f(0) = \mathbf{0}\,. \tag{7.4.1.2c}$$

Hence the approximation is accurate up to polynomials of degree 1, as claimed.

7.4.2 Two points

- Next we study $m = 2$.
- Let us not jump to conclusions what the formula will be.
- ***The answer is not the trapezoid rule.***
- Instead of x_1 and x_2, it is convenient to label the points as x_{-1} and x_1. The two points are

$$x_{\pm 1} = \pm \frac{1}{\sqrt{3}} \,. \tag{7.4.2.1}$$

  The weights $w_{\pm 1}$ are (see eq. (7.2.3))

$$w_{\pm 1} = 1 \,. \tag{7.4.2.2}$$

- Hence the rule is

$$\int_{-1}^{1} f(x)\, dx \simeq f\Big(-\frac{1}{\sqrt{3}}\Big) + f\Big(\frac{1}{\sqrt{3}}\Big) \,. \tag{7.4.2.3}$$

- Let us use this to integrate some polynomials, and compare with the exact result.

$$\int_{-1}^{1} 1\, dx = 2 \,, \qquad f(x_{-1}) + f(x_1) = 1 + 1 \qquad = 2 \,, \tag{7.4.2.4a}$$

$$\int_{-1}^{1} x\, dx = 0 \,, \qquad f(x_{-1}) + f(x_1) = -\frac{1}{\sqrt{3}} + \frac{1}{\sqrt{3}} \qquad = 0 \,, \tag{7.4.2.4b}$$

$$\int_{-1}^{1} x^2\, dx = \frac{2}{3} \,, \qquad f(x_{-1}) + f(x_1) = \frac{1}{3} + \frac{1}{3} \qquad = \frac{2}{3} \,, \tag{7.4.2.4c}$$

$$\int_{-1}^{1} x^3\, dx = 0 \,, \qquad f(x_{-1}) + f(x_1) = -\frac{1}{3\sqrt{3}} + \frac{1}{3\sqrt{3}} \qquad = 0 \,, \tag{7.4.2.4d}$$

$$\int_{-1}^{1} x^4\, dx = \frac{2}{5} \,, \qquad f(x_{-1}) + f(x_1) = \frac{1}{9} + \frac{1}{9} \qquad = \mathbf{\frac{2}{9}} \,. \tag{7.4.2.4e}$$

  Hence the approximation is accurate up to polynomials of degree 3, as claimed.

7.4.3 Comparison with trapezoid rule

- Note that the Gauss-Legendre quadrature is more accurate than the trapezoid rule.
- The trapezoid rule states that (using a stepsize $h = 2$)

$$\int_{-1}^{1} f(x)\,dx \simeq \frac{h}{2}\big(f(-1) + f(1)\big) = f(-1) + f(1)\,. \tag{7.4.3.1}$$

- Let us integrate polynomials using the trapezoid rule:

$$\int_{-1}^{1} 1\,dx = 2\,, \qquad f(-1) + f(1) = 1 + 1 \qquad = 2\,, \tag{7.4.3.2a}$$

$$\int_{-1}^{1} x\,dx = 0\,, \qquad f(-1) + f(1) = -\frac{1}{2} + \frac{1}{2} \qquad = 0\,, \tag{7.4.3.2b}$$

$$\int_{-1}^{1} x^2\,dx = \frac{2}{3}\,, \qquad f(-1) + f(1) = 1 + 1 \qquad = \mathbf{2}\,. \tag{7.4.3.2c}$$

- The trapezoid rule is only accurate up to polynomials of degree 1 (same as the midpoint rule).

7.5 Questions on Gauss-Legendre quadrature

- ***Why is Gauss-Legendre quadrature with two points more accurate than the trapezoid rule?***
- ***How do we determine the locations of the points $x_{\pm 1}$ and the weights $w_{\pm 1}$?***
- ***More generally, for more points, (and all Gaussian quadrature formulas in general), how do we determine the locations of the points x_i and the weights w_i?***

7.6 Answers about Gauss-Legendre quadrature

- Unfortunately, in these lectures, we cannot answer the above questions on Gauss-Legendre quadrature, or Gaussian quadrature in general.
- To explain the theory requires advanced mathematics which is beyond the level of this class.
- We need to know about families of **orthogonal polynomials.**
- A family of orthogonal polynomials, call then $\mathscr{P}_n(x)$, $n = 0, 1, 2, \ldots$ is a set of polynomials which are **orthogonal with respect to a weight function** $w(x)$**.**
- This means that the "orthogonality relation" is (if the domain of integration is $-1 \le x \le 1$)

$$\int_{-1}^{1} w(x)\mathscr{P}_m(x)\mathscr{P}_n(x)\,dx = \delta_{mn}\,. \tag{7.6.1}$$

  Depending on the weight function $w(x)$, the orthogonality relation could instead be

$$\int_{-\infty}^{\infty} w(x)\mathscr{P}_m(x)\mathscr{P}_n(x)\,dx = \delta_{mn}\,. \tag{7.6.2}$$

- The choice of weight function $w(x)$ determines the set of polynomials $\mathscr{P}_n(x)$.
  1. The **Legendre polynomials**, denoted by $P_n(x)$, are orthogonal with respect to the weight function $w(x) = 1$.
  2. The Legendre polynomials are defined on the interval $-1 \le x \le 1$.
  3. It is conventional to normalize the Legendre polynomials so that $P_n(1) = 1$.
  4. With the above normalization, the orthogonality relation is

$$\int_{-1}^{1} P_m(x)P_n(x)\,dx = \frac{2}{2n+1}\,\delta_{mn}\,. \tag{7.6.3}$$

- The first few Legendre polynomials are

$$P_0(x) = 1\,, \tag{7.6.4a}$$
$$P_1(x) = x\,, \tag{7.6.4b}$$
$$P_2(x) = \tfrac{1}{2}(3x^2 - 1)\,, \tag{7.6.4c}$$
$$P_3(x) = \tfrac{1}{2}(5x^3 - 3x)\,. \tag{7.6.4d}$$

- The Legendre polynomial $P_m(x)$ has m distinct roots. The locations of the points x_i are given by the **roots** of the Legendre polynomial. The weights w_i are given by the formula

$$w_i = \frac{2}{(1 - x_i^2)[P_m'(x_i)]^2}\,. \tag{7.6.5}$$

- You can see that the theory to justify all this is complicated.

7.7 Gauss-Legendre with three points

- For three points $m = 3$, the locations of the points are given by the roots of $P_3(x)$:

$$x_0 = 0\,, \qquad x_{\pm 1} = \pm\sqrt{\frac{3}{5}}\,. \tag{7.7.1}$$

- The corresponding weights are (the sum of the weights always adds up to 2)

$$w_0 = \frac{8}{9}\,, \qquad w_{\pm 1} = \frac{5}{9}\,. \tag{7.7.2}$$

- The Gauss-Legendre quadrature formula with three points is

$$\int_{-1}^{1} f(x)\,dx \simeq \frac{1}{9}\left[\,5f\left(-\sqrt{\frac{3}{5}}\right) + 8f(0) + 5f\left(\sqrt{\frac{3}{5}}\right)\,\right]. \tag{7.7.3}$$

- This formula yields the exact result for all polynomials up to degree five.

- Simpson's rule also uses three points, and is

$$\int_{-1}^{1} f(x)\,dx \simeq \frac{1}{3}\left(\,f(-1) + 4f(0) + f(1)\,\right). \tag{7.7.4}$$

- Simpson's rule yields the exact result for all polynomials up to degree three.

7.8 Numerical integration in multiple dimensions

7.8.1 General remarks

- The most important fact about the numerical evaluation of multi-dimensional integrals is that the problem is **computationally difficult**.
- There are many complications and we shall list a few.
- Since we have used n to denote the number of subintervals in the one-dimensional algorithms above, to avoid confusion we shall denote the number of dimensions by d.
- We wish to numerically compute the following integral in a d-dimensional domain Ω:

$$\begin{aligned} I &= \int_\Omega f(x_1, x_2, \ldots, x_d)\, dx_1\, dx_2\, \ldots\, dx_d \\ &= \int_\Omega f(\boldsymbol{x})\, d^d\boldsymbol{x}\,. \end{aligned} \tag{7.8.1.1}$$

  (The last expression is just a more concise vector notation.)

7.8.2 Curse of dimensionality

- The **curse of dimensionality** is a **bad thing.**
- For most algorithms, the computational complexity increases greatly as the number of dimensions increases. Here is one reason why. Suppose the domain of integration is a d-dimensional hypercube. We might as well say it is the unit hypercube $[0,1]^d$.
  1. Let us employ the midpoint rule to compute the integral.
  2. Then we subdivide the hypercube into little cells. If there are n subintervals in each dimension, there will be totally n^d cells in the hypercube.
  3. Hence we must compute the function at n^d points, at the center of each little hypercube.
  4. If we set $n = 10$ (which is not a lot), then in d dimensions this equals 10^d points. For $d = 3$ this means 1000 points, for $d = 6$ we require one million points, and for $d = 9$ we require *one billion* points.
  5. Using only $n = 10$ may not yield a very accurate numerical result, yet in 9 dimensions it requires one billion function evaluations. Using $n = 100$ requires $(100)^9 = 10^{18}$ points in 9 dimensions.
- This geometric increase in the number of points required, as the number of dimensions increases, is called the **curse of dimensionality.**
- The curse of dimensionality makes many algorithms for multi-dimensional numerical integration computationally very expensive.

7.8.3 Shape of domain

- In one dimension, the domain of integration is just an interval. The boundary consists of only two end points. (If we have multiple intervals, we just sum the results for each interval.)
- In multiple dimensions, the domain of integration may not have a simple shape.
- Even in two dimensions, the boundary of the domain of integration may be a complicated curve.
- Hence algorithms such as the trapezoid rule, Simpson's rule and quadrature do not work in multi-dimensions. (Obviously also the extended trapezoid rule and Romberg integration do not work.)
- There is a technique called **cubature**, which is a generalization of quadrature, but we shall not discuss it here. Once again we have to find a set of interpolating points, but now in multiple dimensions. The technique also suffers from the curse of dimensionality.
- Hence these lectures will contain only a limited discussion of algorithms for numerical integration in multiple dimensions.

7.8.4 Factorization

- Pay attention to the structure of the integrand (and the shape of the domain).
- For example, ***the integral may factorize into a product***

$$I = \left(\int_{a_1}^{b_1} f_1(x_1)\, dx_1 \right) \left(\int_{a_2}^{b_2} f_2(x_2)\, dx_1 \right) \ldots \left(\int_{a_d}^{b_d} f_d(x_d)\, dx_d \right). \tag{7.8.4.1}$$

  1. *The overall integral is just a product of one-dimensional integrals.*
  2. Each integral can be computed using high-accuracy one-dimensional algorithms.
  3. There is no problem with the curse of dimensionality.

- ***Pay attention!***
- Computing a multi-dimensional integral is difficult.
  See if it (or at least some part of it) can be factorized out of the total.

7.8.5 Nested integrals

- If the number of dimensions is not very large, so that the curse of dimensionality is not serious, and if the shape of the boundary is not too complicated, then one can evaluate the integral using **nested integrals**.

- Consider for example two dimensions (x, y).

  1. Suppose the limits of integration for x are a_x and b_x.
  2. Next, the limits of integration for y are $a_y(x)$ and $b_y(x)$.
     Note that they depend on the value of x.
  3. For example suppose the boundary of the domain is the unit circle $x^2 + y^2 = 1$. Then

$$a_x = -1\,, \qquad b_x = 1\,, \qquad a_y(x) = -\sqrt{1-x^2}\,, \qquad b_y(x) = \sqrt{1-x^2}\,. \tag{7.8.5.1}$$

  4. We compute the integral as follows

$$I = \int_{a_x}^{b_x} dx \left(\int_{a_y(x)}^{b_y(x)} dy\, f(x,y) \right). \tag{7.8.5.2}$$

  5. This is a nested set of one-dimensional integrals.
  6. Each nested integral is computed using a high-accuracy one-dimensional algorithm.
  7. Note that the points at which we evaluate the function **do not have to lie on a rectangular grid.**
  8. For example, because $a_y(x) = -\sqrt{1-x^2}$ and $b_y(x) = \sqrt{1-x^2}$ both depend on x, each time we compute the integral over y, the spacing of the points, etc. can be different, because the locations of the end points are different.

7.8.6 Coordinate transformations

- It is not necessary to always employ a rectangular grid in higher dimensions.
- In one dimension the domain of integration is just a straight line interval.
- In higher dimensions the shape of the domain of integration can have a non-rectangular shape.
- In the previous example, the domain of integration was a disk $x^2 + y^2 \leq 1$.
    1. In such a situation, it might be better to use polar coordinates (r, θ).
    2. The domain in polar coordinates is easy: $0 \leq r \leq 1$ and $0 \leq \theta \leq 2\pi$.
    3. The integral may be easier to compute using polar coordinates.
- Clearly, other coordinate transformations are also possible.

7.8.7 Monte Carlo integration

- **Monte Carlo integration** is a truly multi-dimensional algorithm.
- The term **Monte Carlo** is employed for many applications in mathematics (or numerical analysis) to denote algorithms which employ statistical sampling.
- The name obviously comes from Monte Carlo in Monaco, and its famous casino, i.e. gambling and random numbers.
- Monte Carlo integration works as follows.
  1. Let the domain of integration Ω be contained in the d-dimensional unit hypercube $[0,1]^d$. We can always scale the coordinate axes to make this possible.
  2. We generate a set of N points say $\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_N$ as follows.
  3. For each point $\boldsymbol{x}_i$, we employ a pseudorandom number generator with a uniform distribution in $[0,1]$ to yield the values of the components $x_{i1}, x_{i2}, \ldots, x_{id}$.
  4. Hence the set of N points $\{\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_N\}$ are uniformly distributed in the d-dimensional unit hypercube $[0,1]^d$.
  5. If the point $\boldsymbol{x}_i$ lies in the domain Ω, we compute the function value $f(\boldsymbol{x}_i)$.
  6. We evaluate the integral via a sum over all the points which lie in the domain Ω:
  $$I_{\text{Monte Carlo}} = \frac{1}{N} \sum_{\boldsymbol{x}_i \in \Omega} f(\boldsymbol{x}_i)\,. \tag{7.8.7.1}$$
  7. Note that if $f(\boldsymbol{x}) = 1$, the above sum yields the (approximate) volume of the domain Ω. This by itself can be an important computational calculation.
- Monte Carlo integration avoids the curse of dimensionality.
- Obviously the numerical result is approximate not only because the value of N is finite, but it is also approximate because the set of points is a ***statistical sample.*** Using a different set of N points will in general yield a different numerical result. This was not the case with the algorithms we studied previously: the set of points was deterministic.
- Monte Carlo integration is an example of a **non-deterministic algorithm**.
- From the theory of probability and statistics, the accuracy of Monte Carlo integration is
  $$I_{\text{Monte Carlo}} = I_{\text{exact}} + O\Big(\frac{1}{\sqrt{N}}\Big)\,. \tag{7.8.7.2}$$
  This is clearly a much slower rate of convergence than, for example, Simpson's rule.
- The slow rate of convergence of the numerical accuracy of Monte Carlo integration is a drawback of the algorithm, but Monte Carlo integration avoids the curse of dimensionality.
- Furthermore, at a later date we can generate a fresh random sample of points (not necessarily N points, it could be $2N$ or $100N$, etc.) and combine all the random samples into one large sample. Monte Carlo integration allows this, which other algorithms do not.

7.8.8 Pseudorandom number generators: a word of caution

- Creating a good quality pseudorandom number generator (PRNG) is **very difficult.**
- The creation of good quality pseudorandom number generators is a vast topic of numerical analysis in its own right.

7.8.9 Quasi-random algorithms

- There are alternative numerical algorithms for multi-dimensional numerical integration, which are called **quasi-random algorithms**.
- The term "quasi-random" is a misnomer, because they are deterministic algorithms.
- We shall not discuss any explicit examples of quasi-random algorithms, just the general idea.
- The motivation is to improve on the slow $O(1/\sqrt{N})$ rate of convergence of Monte Carlo integration, but to avoid the curse of dimensionality.
- The basic idea is to generate a deterministic set of N points, where $N < n^d$, which are uniformly distributed in the unit hypercube $[0, 1]^d$.
- In fact, we want $N \ll n^d$, to avoid the curse of dimensionality.
- Such sets of points go by the general name of **low discrepancy sequences.**
- There are various examples of low discrepancy sequences.
- The rate of convergence of the quasi-random algorithms is $O(1/N)$:

$$I_{\text{quasi-random}} = I_{\text{exact}} + O\Big(\frac{1}{N}\Big)\,. \tag{7.8.9.1}$$

- There are various analyses of low discrepancy sequences which state that a more accurate estimate is

$$I_{\text{quasi-random}} = I_{\text{exact}} + O\Big(\frac{(\ln N)^d}{N}\Big)\,. \tag{7.8.9.2}$$

- Because the set of N points is deterministic, we cannot combine data sets.
- For example, if we wish to use a set of $N + 1$ points, we must generate a whole new set of $N + 1$ points and perform the calculation from the beginning. We cannot simply add one extra point to the previous dataset of N points.
- This is a drawback of the quasi-random algorithms.
  Monte Carlo integration offers a great advantage in this respect.

7.9 Monte Carlo integration: worked example

7.9.1 General remarks

- The domain of integration Ω is the d-dimensional unit ball, given by $\boldsymbol{x} \cdot \boldsymbol{x} \leq 1$, where $\boldsymbol{x}$ is a point in d-dimensional space.
- Let us compute the volume of the unit ball in d dimensions, say V_d

$$V_d = \int_{\boldsymbol{x}\cdot\boldsymbol{x}\leq 1} d^d\boldsymbol{x}\,. \tag{7.9.1.1}$$

- For fun, let us also compute the following integral in the same domain (unit ball)

$$F_d = \frac{1}{V_d}\int_{\boldsymbol{x}\cdot\boldsymbol{x}\leq 1} e^{-\boldsymbol{x}\cdot\boldsymbol{x}/2}\, d^d\boldsymbol{x}\,. \tag{7.9.1.2}$$

- The values of both V_d and F_d can be calculated exactly.
  For simplicity, let d be even $d = 2, 4, 6, \ldots$.
  Then

$$V_d = \frac{\pi^{d/2}}{(d/2)!}\,. \tag{7.9.1.3}$$

  We can define $V_0 = 1$. Then there is a recurrence

$$V_d = \frac{\pi}{d/2}\, V_{d-2} \qquad\qquad (d = 2, 4, 6, \ldots)\,. \tag{7.9.1.4}$$

  The value of F_d can be obtained by solving the following recurrence, starting from $F_0 = 1$:

$$F_d = d\big(F_{d-2} - e^{-1/2}\big) \qquad\qquad (d = 2, 4, 6, \ldots)\,. \tag{7.9.1.5}$$

- The values of V_d and F_d can also be calculated exactly for odd d, but the formulas are more complicated. We shall consider only an even number of dimensions below.
- Let us analyze the structures of the integrals in both eqs. (7.9.1.1) and (7.9.1.2).
  1. Both integrands depend only on the amplitude square $\boldsymbol{x} \cdot \boldsymbol{x}$.
  2. Also in both cases the domain is the unit ball, which is symmetric around the origin in all dimensions.
  3. Hence we *could* transform to polar coordinates.
  4. However, the purpose of this exercise is to demonstrate Monre Carlo integration, using a simple example where the result can be checked against the exact value.
  5. However, we can restrict the domain of integration to the region where all the coordinates are positive

$$0 \leq x_1 \leq 1\,, \quad 0 \leq x_2 \leq 1\,, \quad \ldots \quad 0 \leq x_d \leq 1\,, \qquad \sum_{i=1}^{d} x_i^2 \leq 1\,. \tag{7.9.1.6}$$

  6. Then just multiply the output of the Monte Carlo integration by 2^d to calculate V_d.
  7. The numerator integral in eq. (7.9.1.2) is divided by V_d so no factor of 2^d is required to calculate F_d.

7.9.2 Monte Carlo integration: worked example (function output)

- A working C++ function code to compute the values of V_d and F_d via Monte Carlo integration is given below.
- The function has return type `int` to return 1 (fail) for bad inputs and 0 for success.
- An initial seed value is required for the pseudorandom number generator. The seed is set internally in the function. In a better implementation, the calling application would supply an input value for the seed.
- The program outputs for V_d and F_d are tabulated in Table 7.9.2 below, for $d = 10$.
- The number of points N was run in a loop from 10^5 to 10^6, in steps of 10^5.
- The relative differences are also tabulated in Table 7.9.2

$$\Delta V_d = \frac{|V_{d\,\mathrm{MC}} - V_{d\,\mathrm{exact}}|}{V_{d\,\mathrm{exact}}}\,, \qquad \Delta F_d = \frac{|F_{d\,\mathrm{MC}} - F_{d\,\mathrm{exact}}|}{F_{d\,\mathrm{exact}}}\,. \tag{7.9.2.7}$$

- Notice that in this example, the relative accuracy for V_d decreased by only a facor of about 1.82, as compared to $\sqrt{10^6/10^5} = \sqrt{10} \simeq 3.162$.
- On the other hand, the relative accuracy for F_d decreased by a facor of about 26.3, which is much more than $\sqrt{10} \simeq 3.162$.
- For both V_d and F_d, the accuracy ***does not improve monotonically as the value of N increases.***
- **Always bear in mind that Monte Carlo involves *statistical sampling,* and the accuracy is not guaranteed to increase monotonically with N.**

N	V_d (exact)	V_d (MC)	ΔV_d	F_d (exact)	F_d (MC)	ΔF_d
100000	2.55016	2.65216	0.0399958	0.660924	0.657364	0.00538607
200000	2.55016	2.74944	0.0781424	0.660924	0.660109	0.00123343
300000	2.55016	2.59755	0.0185802	0.660924	0.660604	0.000484522
400000	2.55016	2.57536	0.00988013	0.660924	0.659887	0.00156973
500000	2.55016	2.54771	0.000961522	0.660924	0.660562	0.000547505
600000	2.55016	2.53099	0.00752005	0.660924	0.659342	0.00239424
700000	2.55016	2.50587	0.0173674	0.660924	0.659246	0.0025395
800000	2.55016	2.5408	0.00367194	0.660924	0.659091	0.00277346
900000	2.55016	2.53952	0.00417386	0.660924	0.659499	0.00215686
1000000	2.55016	2.60608	0.0219264	0.660924	0.661059	0.000204699

Table 1: Table of outputs for Monte Carlo integration for V_d and F_d, for $d = 10$, for values $N = 10^5$ to 10^6.

7.9.3 Monte Carlo integration: worked example (code)

```
int Monte_Carlo_unitball(int d, int N, double & V, double & F)
{
  V = 0;
  F = 0;
  if ((d < 1) || (N < 1)) {   // validation checks
    return 1;   // fail
  }

  long iseed = 0;
  while (iseed == 0) {
    iseed = (long) 1234*time(0);  // crude algorithm to set random initial seed
    if (iseed < 0) iseed = -iseed;
  }

  std::default_random_engine generator;
  generator.seed( iseed );
  std::uniform_real_distribution<double> unif_distribution(0.0, 1.0);

  double x[d];  // not really needed

  long i = 0;
  long j = 0;
  double sumV = 0;
  double sumF = 0;
  for (i = 0; i < N; ++i) {
    double r_square = 0;
    for (j = 0; j < d; ++j) {
      double rnd = unif_distribution(generator);   // generate random coordinates
      x[j] = rnd;   // not really needed
      r_square += rnd*rnd;
    }
    if (r_square <= 1.0) {        // test if point lies in domain (or on boundary)
      sumV += 1.0;
      sumF += exp(-0.5*r_square);
    }
  }
  V = sumV * pow(2.0,d)/double(N);
  F = sumF / sumV;

  return 0;   // ok
}
```