Chapter 1

Numerical integration methods

The ability to calculate integrals is quite important. The author was told that, in the old days, the gun ports were cut into a ship only after it was afloat, loaded with equivalent cargo, and rigged. This is because it was impossible to calculate the water displaced volume, i.e. the integral of the hull-describing function, with the rudimentary math known at that time. Consequently, there was no way to properly estimate the buoyant force. Thus, the location of the waterline was unknown until the ship was fully afloat.

Additionally, not every integral can be computed analytically, even for relatively simple functions.

Example

The Gauss error function defined as

$$\operatorname{erf}(y) = \frac{2}{\sqrt{\pi}} \int_0^y e^{-x^2} dx$$

cannot be calculated using only elementary functions.

1.1 Integration problem statement

At first, we consider the evaluation of a one dimensional integral, also called *quadrature*, since this operation is equivalent to finding of the area under the curve of a given function.

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

Surprisingly, one does not need to know any high level math to do so. All you need is a precision scale and scissors. The recipe goes like this: plot the function on some preferably

dense and heavy material, cut the area of interest with scissors, measure the mass of the cutout, divide the obtained mass by the density and thickness of the material and you are done. Well, this is, of course, not very precise and sounds so low tech. We will employ more modern numerical methods.

Words of wisdom

Once you become proficient with computers, there is a tendency to do every problem numerically. Resist it! If you can find an analytical solution for a problem, do so. It is usually faster to calculate, and more importantly, will provide you some physical intuition for the problem overall. Numerical solutions usually do not possess predictive power.

1.2 The Rectangle method

The definition of the integral via Riemann's sum:

$$\int_a^b f(x)dx = \lim_{N\to\infty} \sum_{i=1}^{N-1} f(x_i)(x_{i+1} - x_i) \tag{1.1}$$

where $a \leq x_i \leq b$ and N is the number of points,

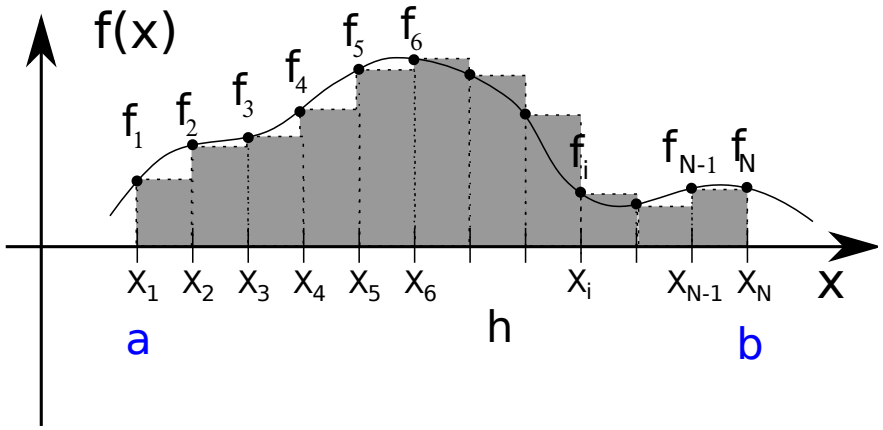

Figure 1.1: The rectangle method illustration. Shaded boxes show the approximations of the area under the curve. Here, $f_1 = f(x_1 = a), f_2 = f(x_2), f_3 = f(x_3), \ldots, f_N = f(x_N = b)$.

Riemann's definition gives us directions for the rectangle method: approximate the area under the curve by a set of boxes or rectangles (see fig. 1.1). For simplicity, we evaluate our function at N points equally separated by the distance h on the interval (a, b).

Rectangle method integral estimate

$$\int_a^b f(x)dx \approx \sum_{i=1}^{N-1} f(x_i)h, \tag{1.2}$$

where

$$h = \frac{b-a}{N-1}, \quad x_i = a + (i-1)h, \quad x_1 = a \quad \text{and} \quad x_N = b \tag{1.3}$$

The MATLAB implementation of this method is quite simple:

Listing 1.1: integrate_in_1d.m (available at http://physics.wm.edu/programming_with_MATLAB_book/ch_integration/code/integrate_in_1d.m)

```
function integral1d = integrate_in_1d(f, a, b)
% integration with simple rectangle/box method
% int_a^b f(x) dx

N=100; % number of points in the sum
x=linspace(a,b,N);

s=0;
for xi=x(1:end-1) % we need to exclude x(end)=b
    s = s + f(xi);
end

%% now we calculate the integral
integral1d = s*(b-a)/(N-1);
```

To demonstrate how to use it, let's check $\int_0^1 x^2 dx = 1/3$

```
f = @(x) x.^2;
integrate_in_1d(f,0,1)
ans =  0.32830
```

Well, 0.32830 is quite close to the expected value of 1/3. The small deviation from the exact result is due to the relatively small number of points; we used $N = 100$.

If you have previous experience in low level languages (from the array functions implementation point of view) like C or Java, the above implementation is the first thing that comes to your mind. While it is correct, it does not use MATLAB's ability to use matrices as arguments of a function. A better way, which avoids using a loop, is shown below.

Listing 1.2: integrate_in_1d_matlab_way.m (available at http://physics.wm.edu/programming_with_MATLAB_book/ch_integration/code/integrate_in_1d_matlab_way.

m)

```
function integral1d = integrate_in_1d_matlab_way(f, a, b)
% integration with simple rectangle/box method
% int_a^b f(x) dx

N=100; % number of points in the sum
x=linspace(a,b,N);

% if function f can work with vector argument then we can do
integral1d = (b-a)/(N-1)*sum( f( x(1:end-1) ) ); % we exclude x(end)=b
```

Words of wisdom

In MATLAB, loops are generally slower compared to the equivalent evaluation of a function with a vector or matrix argument. Try to avoid loops which iterate evaluation over matrix elements.

1.2.1 Rectangle method algorithmic error

While the rectangle method is very simple to understand and implement, it has awful algorithmic error and slow convergence. Let's see why this is so. A closer look at fig. 1.1 reveals that a box often underestimates (see, for example, the box between x_1 and x_2) or overestimates (for example, the box between x_6 and x_7) the area. This is the algorithmic error which is proportional to $f'h^2/2$ to the first order, i.e. the area of a triangle between a box and the curve. Since we have $N-1$ intervals, this error accumulates in the worst case scenario. So we can say that the algorithmic error (E) is

Rectangle method algorithmic error estimate

$$E = \mathcal{O}\left((N-1)\frac{h^2}{2}f'\right) = \mathcal{O}\left(\frac{(b-a)^2}{2N}f'\right) \tag{1.4}$$

In the last term, we replaced $N-1$ with N under the assumption that N is large. The $\mathcal{O}$ symbol represents the *big O notation*, i.e. there is an unknown proportionality coefficient.

1.3 Trapezoidal method

The approximation of each interval with a trapezoid (see fig. 1.2) is an attempt to circumvent the weakness of the rectangle method. In other words, we do a linear approximation of the

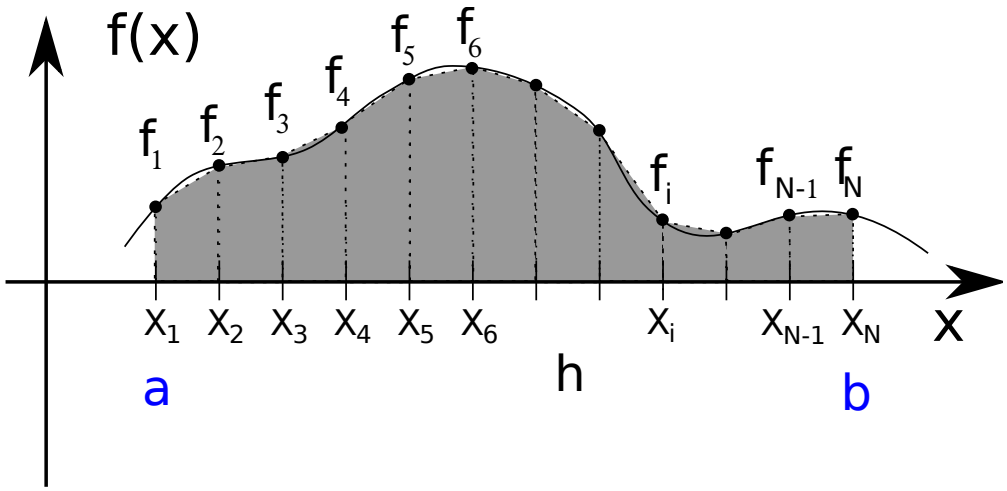

Figure 1.2: The illustration of the trapezoidal method. Shaded trapezoids show the approximations of the area under the curve.

integrated function. Recalling the formula for the area of a trapezoid, we approximate the area of each interval as $h(f_{i+1} + f_i)/2$, and then we sum them all.

Trapezoidal method integral estimate

$$\int_a^b f(x)dx \approx h \times (\frac{1}{2}f_1 + f_2 + f_3 + \cdots + f_{N-2} + f_{N-1} + \frac{1}{2}f_N) \tag{1.5}$$

The 1/2 coefficient disappears for inner points, since they are counted twice: once for the left and once for the right trapezoid area.

Trapezoidal method algorithmic error

To evaluate the algorithmic error, one should note that we are using a linear approximation for the function and ignoring the second order term. Recalling the Taylor expansion, this means that to the first order, we are ignoring contribution of the second derivative (f'') terms. With a bit of patience, it can be shown that

Trapezoidal method algorithmic error estimate

$$E = \mathcal{O}\left(\frac{(b-a)^3}{12N^2}f''\right) \tag{1.6}$$

Let's compare the integral estimate with rectangle (eq. (1.2)) and trapezoidal (eq. (1.5))

methods.

$$\int_a^b f(x)dx \approx h \times (f_2 + f_3 + \cdots + f_{N-2} + f_{N-1}) + h \times (f_1), \tag{1.7}$$

$$\int_a^b f(x)dx \approx h \times (f_2 + f_3 + \cdots + f_{N-2} + f_{N-1}) + h \times \frac{1}{2}(f_1 + f_N) \tag{1.8}$$

It is easy to see that they are almost identical, with the only difference in the second term of either $h \times (f_1)$ for the rectangle method or $h \times \frac{1}{2}(f_1 + f_N)$ for the trapezoidal one. While this might seem like a minor change in the underlying formula, it results in the algorithmic error decreasing as $1/N^2$ for the trapezoidal method, which is way better than the $1/N$ dependence for the rectangle method.

1.4 Simpson's method

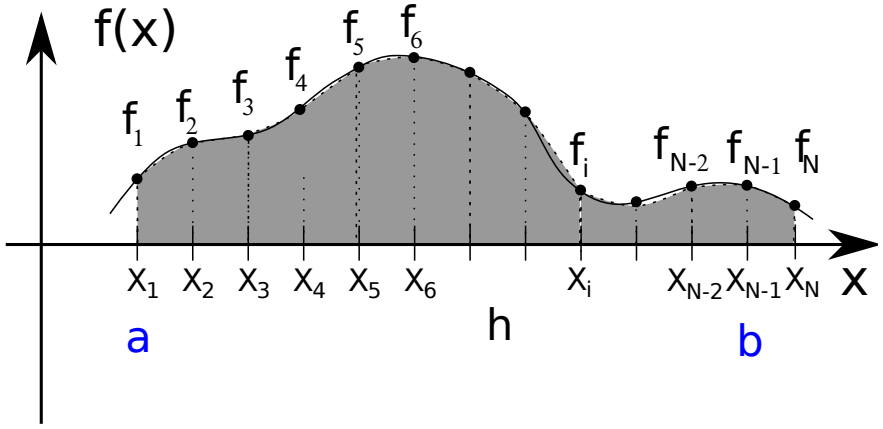

Figure 1.3: Simpson's method illustration

The next logical step is to approximate the function with second order curves, i.e. parabolas (see fig. 1.3). This leads us to the Simpson method. Let's consider a triplet of consequent points (x_{i-1}, f_{i-1}), (x_i, f_i), and (x_{i+1}, f_{i+1}). The area under the parabola passing through all of these points is $h/3 \times (f_{i-1} + 4f_i + f_{i+1})$. Then we sum the areas of all triplets and obtain

Simpson's method integral estimate

$$\int_a^b f(x)dx \approx h\frac{1}{3} \times (f_1 + 4f_2 + 2f_3 + 4f_4 + \cdots + 2f_{N-2} + 4f_{N-1} + f_N) \tag{1.9}$$

N must be in special form $N = 2k + 1$, i.e. odd, and ≥ 3

Yet again, the first (f_1) and last (f_N) points are special, since they are counted only once, while the edge points $f_3, f_5, \ldots$ are counted twice as members of the left and right triplets.

Simpson's method algorithmic error

Since we are using more terms of the function's Taylor expansion, the convergence of the method is improving and the error of the method decreases with N even faster.

Simpson method algorithmic error estimate

$$E = \mathcal{O}\left(\frac{(b-a)^5}{180N^4} f^{(4)}\right) \tag{1.10}$$

Perhaps one would be surprised to see $f^{(4)}$ and N^4. This is because when we integrate a triplet area, we go by the h to the left and to the right of the central point, so the terms proportional to the $x^3 \times f^{(3)}$ are canceled out.

1.5 Generalized formula for integration

A careful reader may have already noticed that the integration formulas for all above methods can be written in the same general form.

Generalized formula for numerical integration

$$\int_a^b f(x)dx \approx h \sum_{i=1}^{N} f(x_i) w_i, \tag{1.11}$$

where w_i is the weight coefficient.

So, one has no excuse to use the rectangle and trapezoid method over Simpson's, since the calculational burden is exactly the same, but the calculation error for Simpson's method drops drastically as the number of points increases.

Strictly speaking, even Simpson's method is not so superior in comparison to others which use even higher order polynomial approximation for the function. These higher order methods would have exactly the same form as eq. (1.11). The difference will be in the weight coefficients. One can see a more detailed discussion of this issue in [1].

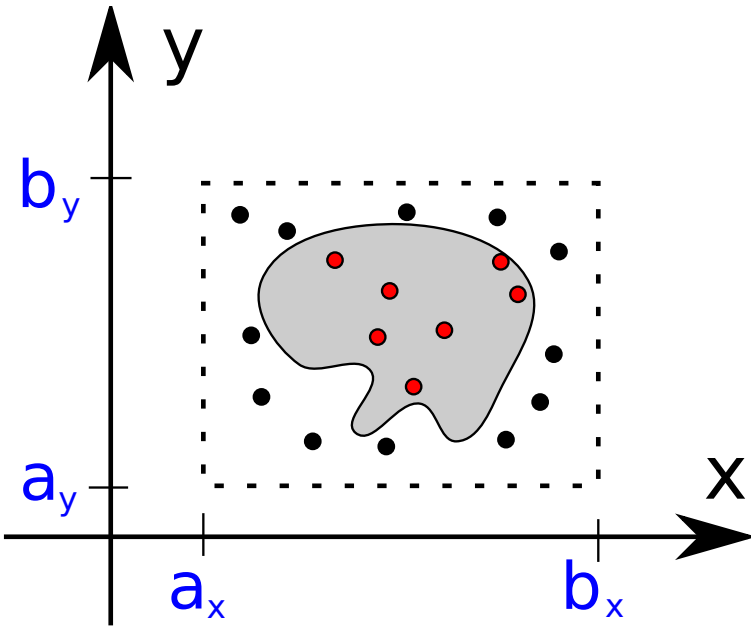

Figure 1.4: The pond area estimate via the Monte Carlo method.

1.6 Monte Carlo Integration

Toy example: finding the area of a pond

Suppose we want to estimate the area of a pond. Naively, we might rush to grab a measuring tape. However, we could just walk around and throw stones in every possible direction, and draw an imaginary box around the pond to see how many stones landed inside (N_{inside}) the pond out of their total number (N_{total}). As illustrated in fig. 1.4, the ratio of above numbers should be proportional to the ratio of the pond area to the box area. So, we conclude that the estimated area of the pond (A_{pond})

$$A_{pond} = \frac{N_{inside}}{N_{total}} A_{box} \tag{1.12}$$

where $A_{box} = (b_x - a_x)(b_y - a_y)$ is the box area. The above estimate requires that the thrown stones are distributed *randomly* and *uniformly*[1]. It is also a good idea to keep the area of the surrounding box as tight as possible to increase the probability of hitting the pond. Imagine what would happen if the box was huge compared to the pond size: we would need a lot of stones (while we have only finite amount of them) to hit the pond even once, and until then the pond area estimate would be zero in accordance with eq. (1.12).

[1] This is not a trivial task. We will talk about this in chapter 11. For now, we will use `rand` function provided by MATLAB.

1.6.1 Naive Monte Carlo integration

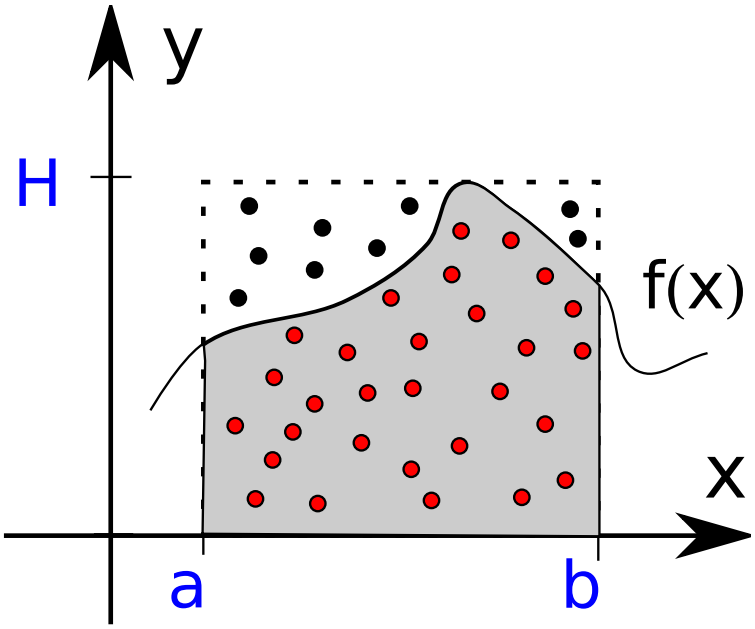

Figure 1.5: Estimate of the integral by counting points under the curve and in the surrounding box.

Well, the calculation of the area under the curve is not much different from the measurement of the pond area as shown in fig. 1.5. So

$$\int_{a_x}^{b_x} f(x)dx = \frac{N_{inside}}{N_{total}} A_{box}$$

1.6.2 Monte Carlo integration derived

The above method is not optimal. Let's focus our attention on a narrow strip around the point x_b (see fig. 1.6). For this strip

$$\frac{N_{inside}}{N_{total}} H \approx f(x_b) \tag{1.13}$$

So, there is no need to waste all of these resources if we can get the $f(x_b)$ right away. Thus, the improved estimate of the integral by the Monte Carlo method looks like

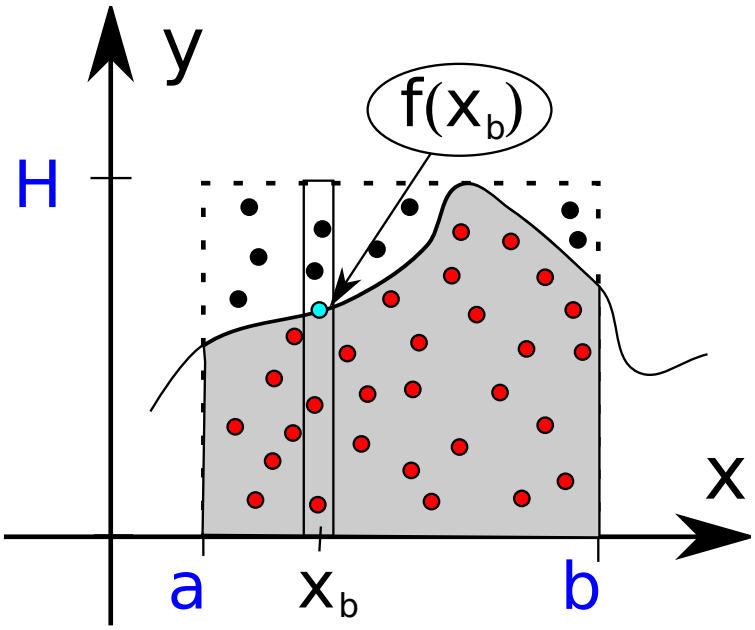

Figure 1.6: Explanation for the improved Monte Carlo method

The Monte Carlo method integral estimate

Choose N uniformly and randomly distributed points x_i inside $[a, b]$

$$\int_a^b f(x)dx \approx \frac{b-a}{N} \sum_{i=1}^{N} f(x_i) \tag{1.14}$$

The Monte Carlo method algorithmic error

A careful look at eq. (1.14) shows that $1/N \sum_{i=1}^{N} f(x_i) =< f >$ is actually a statistical estimate of the function mean. Statistically, we are only sure in this estimate within the standard deviation of the mean estimate. This brings us to the following expression.

Monte Carlo method algorithmic error estimate

$$E = \mathcal{O}\left(\frac{b-a}{\sqrt{N}}\sqrt{< f^2 > - < f >^2}\right) \tag{1.15}$$

where

$$\begin{aligned} < f > &= \frac{1}{N}\sum_{i=1}^{N} f(x_i) \\ < f^2 > &= \frac{1}{N}\sum_{i=1}^{N} f^2(x_i) \end{aligned}$$

The right most term under the square root is the estimate of the function standard deviation $\sigma = \sqrt{< f^2 > - < f >^2}$.

Looking at the above expression, you might conclude that you have wasted precious minutes of your life reading about a very mediocre method, which reduces its error proportionally to $1/\sqrt{N}$. This is worse even compared to the otherwise awful rectangle method.

Hold your horses. In the following section, we will see how the Monte Carlo method outshines all others for the case of multidimensional integration.

1.7 Multidimensional integration

If someone asks us to calculate a multidimensional integral, we just need to apply our knowledge of how to deal with single dimension integrals. For example, in the two dimensional case, we simply rearrange terms:

$$\int_{a_x}^{b_x}\int_{a_y}^{b_y} f(x,y)\ dx\ dy = \int_{a_x}^{b_x} dx \int_{a_y}^{b_y} dy\ f(x,y) \tag{1.16}$$

The last single dimension integral is the function of only x:

$$\int_{a_y}^{b_y} dy\ f(x,y) = F(x) \tag{1.17}$$

So, the two-dimensional integral boils down to the chain of two one-dimension ones, which we are already fit to process:

$$\int_{a_x}^{b_x}\int_{a_y}^{b_y} f(x,y)\ dx\ dy = \int_{a_x}^{b_x} dx\ F(x) \tag{1.18}$$

Minimal example for integration in 2D

The listing below shows how to do 2D integrals by chaining the one-dimension ones; note that it piggybacks on the single dimensional integral in listing 1.1 (but feel free to use any other method).

Listing 1.3: integrate_in_2d.m (available at http://physics.wm.edu/programming_with_MATLAB_book/ch_integration/code/integrate_in_2d.m)

```
function integral2d=integrate_in_2d(f, xrange, yrange)
% Integrates function f in 2D space
% f is handle to function of x, y i.e. f(x,y) should be valid
% xrange is a vector containing lower and upper limits of integration
%   along the first dimension.
%   xrange = [x_lower x_upper]
% yrange is similar but for the second dimension

% We will define (Integral f(x,y) dy) as Fx(x)
Fx = @(x) integrate_in_1d( @(y) f(x,y), yrange(1), yrange(2) );
%   ^^^^^ we fix 'x',      ^^^^^^^^^^^here we reuse this already fixed x
%                                    so it reads as Fy(y)
% This is quite cumbersome.
% It is probably impossible to do a general D-dimensional case.
% Notice that matlab folks implemented integral, integral2, integral3
% but they did not do any for higher than 3 dimensions.

integral2d = integrate_in_1d(Fx, xrange(1), xrange(2) );
end
```

Let's calculate

$$\int_0^2 dx \int_0^1 (2x^2 + y^2)\, dy \tag{1.19}$$

```
f = @(x,y) 2*x.^2 + y.^2;
integrate_in_2d( f, [0,2], [0,1] )
ans =  5.9094
```

It is easy to see that the exact answer is 6. The observed deviation from the analytical result is due to the small number of points used in the calculation.

1.8 Multidimensional integration with Monte Carlo

The above "chain" method can be expanded to any number of dimensions. Can we rest now? Not so fast. Note that if we would like to split the integration region by N points in every of the D dimensions, then the number of evaluations, and thus the calculation time, grows $\sim N^D$. This renders the rectangle, trapezoidal, Simpson's, and alike methods useless for high-dimension integrals.

The Monte Carlo method is a notable exception; it looks very simple even for a multidimensional case, maintains the same $\sim N$ evaluation time, and its error is still $\sim 1/\sqrt{N}$.

A 3D case, for example, would look like this

$$\int_{a_x}^{b_x} dx \int_{a_y}^{b_y} dy \int_{a_z}^{b_z} dz \; f(x,y,z) \approx \frac{(b_x-a_x)(b_y-a_y)(b_z-a_z)}{N} \sum_{i=1}^{N} f(x_i,y_i,z_i) \tag{1.20}$$

and the general form is shown below

Monte Carlo method in D-space

$$\int_{V_D} dV_D \; f(\vec{x}) = \int_{V_D} dx_1 dx_2 dx_3 ... dx_D \; f(\vec{x}) \approx \frac{V_D}{N} \sum_{i=1}^{N} f(\vec{x}_i) \tag{1.21}$$

where V_D is the D-dimensional volume , $\vec{x}_i$ randomly and uniformly distributed points in the volume V_D

Monte Carlo method demonstration

To see how elegant and simple the implementation of the Monte Carlo method can be, we will evaluate the integral in eq. (1.19).

```
f = @(x,y) 2*x.^2 + y.^2;
bx=2; ax=0;
by=1; ay=0;
% first we prepare x and y components of random points
% rand provides uniformly distributed points in the (0,1) interval
N=1000; % Number of random points
x=ax+(bx-ax)*rand(1,N);  % 1 row, N columns
y=ay+(by-ay)*rand(1,N);  % 1 row, N columns

% finally integral evaluation
integral2d = (bx-ax)*(by-ay)/N * sum( f(x,y) )
```

```
integral2d =  6.1706
```

We used only 1000 points and the result is quite close to the analytical value of 6.

1.9 Numerical integration gotchas

Using a very large number of points

Since the algorithmic error of numerical integration methods drops with an increase of N, it is very tempting to increase it. But we must remember about round-off errors and resist this temptation. So, h should not be too small or equivalently N should not be too big. N definitely should not be infinite as the Riemann's sum in eq. (1.1) prescribes, since our lifetime is finite.

Using too few points

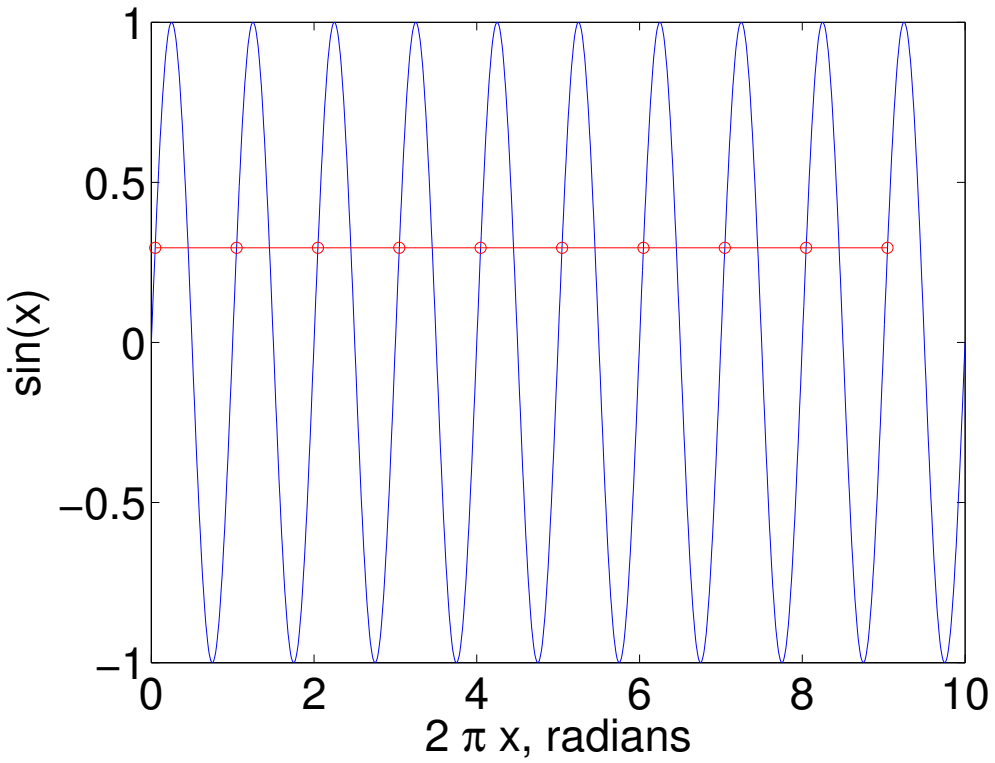

Figure 1.7: The example of badly chosen points ($\circ$) for integration of the $\sin(x)$ function ($-$). The samples give the impression that the function is a horizontal line.

There is a danger of under sampling if the integrated function changes very quickly and we put points very sparsely (see for example fig. 1.7). We should first plot the function (if it is not too computationally taxing), then choose appropriate amount of points: there should be at least 2 and ideally more points for every valley and hill of the function[2]. To see if you hit a sweet spot, try to double the amount of points and see if the estimate of the integral stops changing drastically. This is the foundation for the so called *adaptive integration* method, where an algorithm automatically decides on the required number of points.

1.10 MATLAB functions for integration

Below is a short summary for MATLAB's built-in functions for integration:

1D integration

- `integral`
- `trapz` employs modified trapezoidal method
- `quad` employs modified Simpson's method

Here is how to use `quad` to calculate $\int_0^2 3x^3 dx$

```
>> f = @(x) 3*x.^2;
>> quad(f, 0, 2)
ans =
    8.0000
```

As expected, the answer is 8. Note that `quad` expects your function to be vector friendly.

Multidimensional integration

- `integral2` 2D case
- `integral3` 3D case

Let's calculate the integral $\int_0^1 \int_0^1 (x^2 + y^2) dxdy$ which equals to $2/3$.

```
>> f = @(x,y) x.^2 + y.^2;
>> integral2( f, 0, 1, 0, 1)
ans =
    0.6667
```

There are many others built-ins. See MATLAB's numerical integration documentation to learn more.

[2] We will discuss it in detail in the section devoted to the discrete Fourier transform (chapter 15).

MATLAB's implementations are more powerful than those which we discussed, but deep inside, they use similar methods.

1.11 Self-Study

General comments:

- Do not forget to run some test cases.
- MATLAB has built-in numerical integration methods, such as `quad`. You might check the validity of your implementations with answers produced by this MATLAB built-in function. `quad` **requires your function to be able to work with an array of x points**, otherwise it will fail.
  - Of course, it is always better to compare to the exact analytically calculated value.

Problem 1: Implement the trapezoidal numerical integration method. Call your function `trapezInt(f,a,b,N)`, where a and b are left and right limits of integration, N is the number of points, and f is handle to the function.

Problem 2: Implement the Simpson numerical integration method. Call your function `simpsonInt(f,a,b,N)`. Remember about special form of N=2k+1.

Problem 3: Implement the Monte-Carlo numerical integration method. Call your function `montecarloInt(f,a,b,N)`.

Problem 4: For your tests calculate

$$\int_0^{10} [\exp(-x) + (x/1000)^3]dx$$

Plot the integral absolute error from its true value for the above methods (include rectangular method as well) vs. different number of points N. Try to do it from small N=3 to N=10^6. Use `loglog` plotting function for better representation (make sure that you have enough points in all areas of the plot). Can you relate the trends to eqs. (1.4), (1.6), (1.9) and (1.15). Why does the error start to grow with a larger N? Does it grow for all methods? Why?

Problem 5: Calculate

$$\int_0^{\pi/2} \sin(401x)dx$$

Compare your result with the exact answer 1/401. Provide a discussion about required number of points to calculate this integral.

Problem 6: Calculate

$$\int_{-1}^{1} dx \int_{0}^{1} dy(x^4 + y^2 + x\sqrt{y})$$

Compare results of the Monte-Carlo numerical integration method and MATLAB's built-in `integral2`.

Problem 7: Implement a method to find the volume of the N-dimensional sphere for the arbitrary dimension N and the sphere radius R

$$V(N, R) = \iiint_{x_1^2+x_2^2+x_3^2+\cdots+x_N^2 \leq R^2} dx_1 dx_2 dx_3 \cdots dx_N$$

Calculate the volume of the sphere with $R = 1$ and $N = 20$.

Bibliography

[1] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes 3rd Edition: The Art of Scientific Computing*. Cambridge University Press, New York, NY, USA, 3 edition, 2007.