Numerical Integration

Aksel Hiorth, the National IOR Centre & Institute for Energy Resources,

University of Stavanger

Nov 22, 2019

Contents

1 Numerical Integration

Before diving into the details of this section, it is worth pointing out that the derivation of the algorithms in this section follows a general pattern:

1. We start with a mathematical model (in this case an integral)
2. The mathematical model is formulated in discrete form

3. Then we design an algorithm to solve the model

4. The numerical solution for a test case is compared with the true solution (could be an analytical solution or data)

5. Error analysis: we investigate the accuracy of the algorithm by changing the number of iterations and/or make changes to the implementation or algorithm

In practice you would not use your own implementation to calculate an integral, but in order to understand which method to use in a specific case, it is important to understand the limitation and advantages of the different algorithms. The only way to achieve this is to have a basic understanding of the development. There might also be some cases where you would like to adapt an integration scheme to your specific case if there is a special need that the integration is fast.

1.1 The Midpoint Rule

Numerical integration is encountered in numerous applications in physics and engineering sciences. Let us first consider the most simple case, a function $f(x)$, which is a function of one variable, x. The most straight forward way of calculating the area $\int_a^b f(x)dx$ is simply to divide the area under the function into N equal rectangular slices with size $h = (b-a)/N$, as illustrated in figure 1. The area of one box is:

$$M(x_k, x_k + h) = f(x_k + \frac{h}{2})h, \tag{1}$$

and the area of all the boxes is:

$$\begin{aligned} I(a,b) = \int_a^b f(x)dx &\simeq \sum_{k=0}^{N-1} M(x_k, x_k + h) \\ &= h\sum_{k=0}^{N-1} f(x_k + \frac{h}{2}) = h\sum_{k=0}^{N-1} f(a + (k + \frac{1}{2})h). \end{aligned} \tag{2}$$

Note that the sum goes from $k = 0, 1, \ldots, N-1$, a total of N elements. We could have chosen to let the sum go from $k = 1, 2, \ldots, N$. In Python, C, C++ and many other programming languages the arrays start by indexing the elements from $0, 1, \ldots$ to $N-1$, therefore we choose the convention of having the first element to start at $k = 0$.

Below is a Python code, where this algorithm is implemented for $\int_0^\pi \sin(x)dx$

```
import numpy as np
# Function to be integrated
def f(x):
    return np.sin(x)

def int_midpoint(lower_limit, upper_limit,func,N):
```

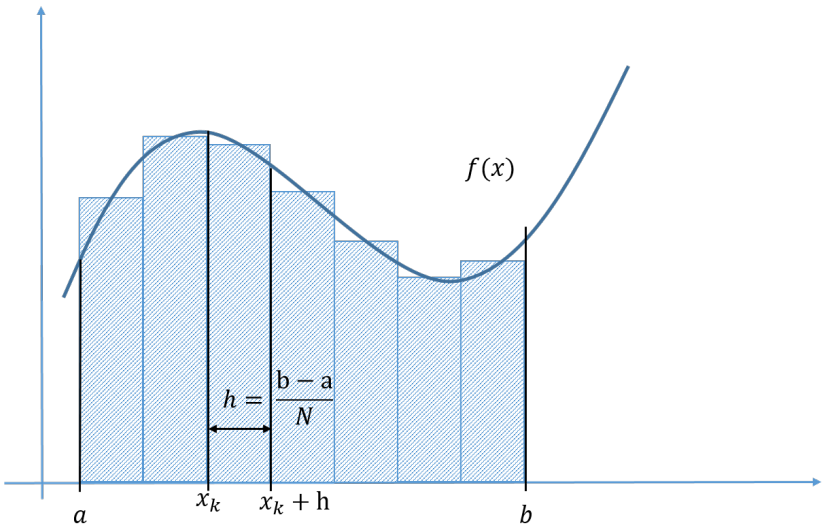

Figure 1: Integrating a function with the midpoint rule.

```
    """ calculates the area of func over the domain lower_limit
        to upper limit using N integration points """
    h    = (upper_limit-lower_limit)/N
    area = 0.
    for k in range(0,N): # loop over k=0,1,..,N-1
        val = lower_limit+(k+0.5)*h # midpoint value
        area += func(val)*h
    return area
N
a=0
b=np.pi
Area = int_midpoint(a,b,f,N)
print('Numerical value= ', Area)
print('Error= ', (2-Area)/2) # Analytical result is 2
```

Notice.

There are many ways to calculate loops in a programming language. If you were coding in a lower level programming language like Fortran, C or C++, you would probably implement the loop like (in Python syntax):

```
for k in range(0,N): # loop over k=0,1,..,N-1
    val = lower_limit+(k+0.5)*h # midpoint value
    area += func(val)
return area*h
```

However, in Python, you would always try to avoid loops because they are generally slow. A more efficient way of implementing the above rule would be to replace the loop with:

```
val  = [lower_limit+(k+0.5)*h for k in range(N)]
ff   = func(val)
area = np.sum(ff)
return area*h
```

1.2 The Trapezoidal Rule

The numerical error in the above example is quite low, only about 2% for $N = 5$. However, by just looking at the graph above it seems likely that we can develop a better algorithm by using trapezoids instead of rectangles, see figure 2.

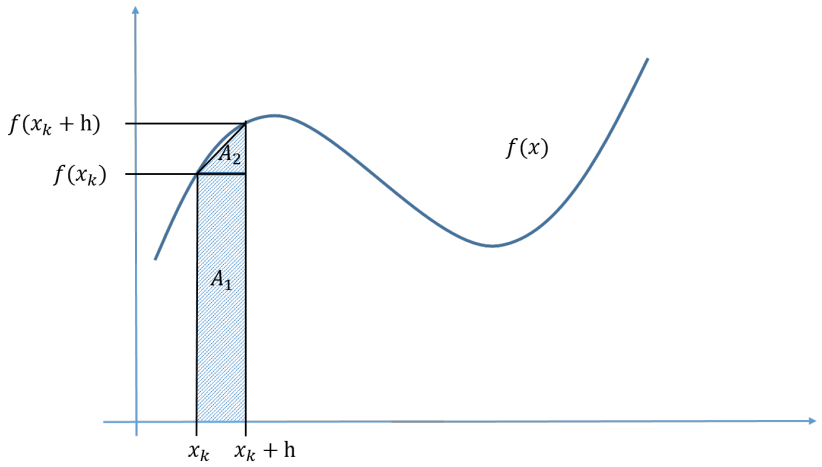

Figure 2: Integrating a function with the trapezoidal rule.

Earlier we approximated the area using the midpoint value: $f(x_k + h/2) \cdot h$. Now we use $A = A_1 + A_2$, where $A_1 = f(x_k) \cdot h$ and $A_2 = (f(x_k + h) - f(x_k)) \cdot h/2$, hence the area of one trapezoid is:

$$A \equiv T(x_k, x_k + h) = (f(x_k + h) + f(x_k))h/2. \tag{3}$$

This is the trapezoidal rule, and for the whole interval we get:

$$
\begin{aligned}
I(a,b) = \int_a^b f(x)dx &\simeq \frac{1}{2}h\sum_{k=0}^{N-1}\left[f(x_k+h)+f(x_k)\right] \\
&= h\left[\frac{1}{2}f(a)+f(a+h)+f(a+2h)+\right. \\
&\qquad \left.\cdots+f(a+(N-2)h)+\frac{1}{2}f(b)\right] \\
&= h\left[\frac{1}{2}f(a)+\frac{1}{2}f(b)+\sum_{k=1}^{N-1}f(a+kh)\right]. \qquad (4)
\end{aligned}
$$

Note that this formula was bit more involved to derive, but it requires only one more function evaluations compared to the midpoint rule. Below is a python implementation:

```
import numpy as np
# Function to be integrated
def f(x):
    return np.sin(x)

#In the implementation below the calculation goes faster
#when we avoid unnecessary multiplications by h in the loop
def int_trapez(lower_limit, upper_limit,func,N):
    """ calculates the area of func over the domain lower_limit
        to upper limit using N integration points """
    h       = (upper_limit-lower_limit)/N
    area    = func(lower_limit)+func(upper_limit)
    area   *= 0.5
    val     = lower_limit
    for k in range(1,N): # loop over k=1,..,N-1
        val  += h # midpoint value
        area += func(val)
    return area*h

N=350
a=0
b=np.pi
Area = int_trapez(a,b,f,N)
print('Numerical value= ', Area)
print('Error= ', (2-Area)) # Analytical result is 2
print('Theoretical Error', np.pi**2/6/N/N)
```

In the table below, we have calculated the numerical error for various values of N.

N	h	Error Midpoint	Error Trapezoidal
1	3.14	-57%	100%
5	0.628	-1.66%	3.31%
10	0.314	-0.412%	0.824%
100	0.031	-4.11E-3%	8.22E-3%

Note that we get the surprising result that this algorithm performs poorer, a factor of 2 than the midpoint rule. How can this be explained? By just looking at figure 1, we see that the midpoint rule actually over predicts the area from

$[x_k, x_k + h/2]$ and under predicts in the interval $[x_k + h/2, x_{k+1}]$ or vice versa. The net effect is that for many cases the midpoint rule give a slightly better performance than the trapezoidal rule. In the next section we will investigate this more formally.

1.3 Numerical Errors on Integrals

It is important to know the accuracy of the methods we are using, otherwise we do not know if the computer produce correct results. In the previous examples we were able to estimate the error because we knew the analytical result. However, if we know the analytical result there is no reason to use the computer to calculate the result(!). Thus, we need a general method to estimate the error, and let the computer run until a desired accuracy is reached.

In order to analyze the midpoint rule in more detail we approximate the function by a Taylor series at the midpoint between x_k and $x_k + h$:

$$\begin{aligned} f(x) &= f(x_k + h/2) + f'(x_k + h/2)(x - (x_k + h/2)) \\ &+ \frac{1}{2!} f''(x_k + h/2)(x - (x_k + h/2))^2 + \mathcal{O}(h^3) \end{aligned} \tag{5}$$

Since $f(x_k + h/2)$ and its derivatives are constants it is straight forward to integrate $f(x)$:

$$\begin{aligned} I(x_k, x_k + h) = \int_{x_k}^{x_k+h} &[f(x_k + h/2) + f'(x_k + h/2)(x - (x_k + h/2)) \\ &+ \frac{1}{2!} f''(x_k + h/2)(x - (x_k + h/2))^2 + \mathcal{O}(h^3)\Big] \, dx \end{aligned} \tag{6}$$

The first term is simply the midpoint rule, to evaluate the two other terms we make the substitution: $u = x - x_k$:

$$\begin{aligned} I(x_k, x_k + h) &= f(x_k + h/2) \cdot h + f'(x_k + h/2) \int_0^h (u - h/2) du \\ &+ \frac{1}{2} f''(x_k + h/2) \int_0^h (u - h/2)^2 du + \mathcal{O}(h^4) \\ &= f(x_k + h/2) \cdot h - \frac{h^3}{24} f''(x_k + h/2) + \mathcal{O}(h^4). \end{aligned} \tag{7}$$

Note that all the odd terms cancels out, i.e $\int_0^h (u - h/2)^m = 0$ for $m = 1, 3, 5 \ldots$. Thus the error for the midpoint rule, $E_{M,k}$, on this particular interval is:

$$E_{M,k} = I(x_k, x_k + h) - f(x_k + h/2) \cdot h = -\frac{h^3}{24} f''(x_k + h/2), \tag{8}$$

where we have ignored higher order terms. We can easily sum up the error on all the intervals, but clearly $f''(x_k + h/2)$ will not, in general, have the same

value on all intervals. However, an upper bound for the error can be found by replacing $f''(x_k + h/2)$ with the maximal value on the interval $[a, b]$, $f''(\eta)$:

$$E_M = \sum_{k=0}^{N-1} E_{M,k} = -\frac{h^3}{24} \sum_{k=0}^{N-1} f''(x_k + h/2) \leq -\frac{Nh^3}{24} f''(\eta), \tag{9}$$

$$E_M \leq -\frac{(b-a)^3}{24N^2} f''(\eta), \tag{10}$$

where we have used $h = (b - a)/N$. We can do the exact same analysis for the trapezoidal rule, but then we expand the function around $x_k - h$ instead of the midpoint. The error term is then:

$$E_T = \frac{(b-a)^3}{12N^2} f''(\overline{\eta}). \tag{11}$$

At the first glance it might look like the midpoint rule always is better than the trapezoidal rule, but note that the second derivative is evaluated in different points (η and $\overline{\eta}$). Thus it is possible to construct examples where the midpoint rule performs poorer than the trapezoidal rule.

Before we end this section we will rewrite the error terms in a more useful form as it is not so easy to evaluate $f''(\eta)$ (since we do not know which value of η to use). By taking a closer look at equation (9), we see that it is closely related to the midpoint rule for $\int_a^b f''(x)dx$, hence:

$$E_M = -\frac{h^2}{24} h \sum_{k=0}^{N-1} f''(x_k + h/2) \simeq -\frac{h^2}{24} \int_a^b f''(x)dx \tag{12}$$

$$E_M \simeq \frac{h^2}{24} \left[f'(b) - f'(a)\right] = -\frac{(b-a)^2}{24N^2} \left[f'(b) - f'(a)\right] \tag{13}$$

The corresponding formula for the trapezoid formula is:

$$E_T \simeq \frac{h^2}{12} \left[f'(b) - f'(a)\right] = \frac{(b-a)^2}{12N^2} \left[f'(b) - f'(a)\right] \tag{14}$$

Now, we can make an algorithm that automatically choose the number of steps to reach (at least) a predefined accuracy:

```
import numpy as np
# Function to be integrated
def f(x):
    return np.sin(x)
#Numerical derivative of function
def df(x,func):
    dh=1e-5 # some low step size
    return (func(x+dh)-func(x))/dh

#Adaptive midpoint rule, "adaptive" because the number of
#function evaluations depends on the integrand
def int_adaptive_midpoint(lower_limit, upper_limit,func,tol):
    dfa  = df(lower_limit,func) # derivative in point a
```

```
    dfb  = df(upper_limit,func) # derivative in point b
    N    = abs((upper_limit-lower_limit)**2*(dfb-dfa)/24/tol)
    N    = int(np.sqrt(N)) + 1  # +1 as int rounds down
    h    = (upper_limit-lower_limit)/N
    area = 0.
    print('Number of intervals = ', N)
    for k in range(0,N): # loop over k=0,1,..,N-1
        val = lower_limit+(k+0.5)*h # midpoint value
        area += func(val)
    return area*h

prec=1e-4
a=0
b=np.pi
Area = int_adaptive_midpoint(a,b,f,prec)
print('Numerical value = ', Area)
print('Error           = ', (2-Area)) # Analytical result is 2
```

Notice.

In Python it is sometimes convenient to enter default values for the arguments in a function. In the above example, we could also have written the function definition as
`def int_adaptive_midpoint(func, lower_limit, upper_limit, tol=1e-8):`. If the `tol` parameter is not given the code will assume an accuracy of 10^{-8}.

1.4 Practical Estimation of Errors on Integrals (Richardson Extrapolation)

From the example above we were able to estimate the number of steps needed to reach (at least) a certain precision. In many practical cases we do not deal with functions, but with data and it can be difficult to evaluate the derivative. We also saw from the example above that the algorithm gives a higher precision than what we asked for. How can we avoid doing too many iterations? A very simple solution to this question is to double the number of intervals until a desired accuracy is reached. The following analysis holds for both the trapezoid and midpoint method, because in both cases the error scale as h^2.

Assume that we have evaluated the integral with a step size h_1, and the computed result is I_1. Then we know that the true integral is $I = I_1 + ch_1^2$, where c is a constant that is unknown. If we now half the step size: $h_2 = h_1/2$, then we get a new (better) estimate of the integral, I_2, which is related to the true integral I as: $I = I_2 + ch_2^2$. Taking the difference between I_2 and I_1 give us an estimation of the error:

$$I_2 - I_1 = I - ch_2^2 - (I - ch_1^2) = 3ch_2^2, \tag{15}$$

where we have used the fact that $h_1 = 2h_2$, Thus the error term is:

$$E(a,b) = ch_2^2 = \frac{1}{3}(I_2 - I_1). \tag{16}$$

This might seem like we need to evaluate the integral twice as many times as needed. This is not the case, by choosing to exactly half the spacing we only need to evaluate for the values that lies halfway between the original points. We will demonstrate how to do this by using the trapezoidal rule, because it operates directly on the x_k values and not the midpoint values. The trapezoidal rule can now be written as:

$$I_2(a,b) = h_2 \left[\frac{1}{2} f(a) + \frac{1}{2} f(b) + \sum_{k=1}^{N_2-1} f(a+kh_2) \right], \tag{17}$$

$$= h_2 \left[\frac{1}{2} f(a) + \frac{1}{2} f(b) + \sum_{k=\text{even values}}^{N_2-1} f(a+kh_2) + \sum_{k=\text{odd values}}^{N_2-1} f(a+kh_2) \right], \tag{18}$$

in the last equation we have split the sum into odd an even values. The sum over the even values can be rewritten:

$$\sum_{k=\text{even values}}^{N_2-1} f(a+kh_2) = \sum_{k=0}^{N_1-1} f(a+2kh_2) = \sum_{k=0}^{N_1-1} f(a+kh_1), \tag{19}$$

note that N_2 is replaced with $N_1 = N_2/2$, we can now rewrite I_2 as:

$$I_2(a,b) = h_2 \left[\frac{1}{2} f(a) + \frac{1}{2} f(b) + \sum_{k=0}^{N_1-1} f(a+kh_1) + \sum_{k=\text{odd values}}^{N_2-1} f(a+kh_2) \right] \tag{20}$$

Note that the first terms are actually the trapezoidal rule for I_1, hence:

$$I_2(a,b) = \frac{1}{2} I_1(a,b) + h_2 \sum_{k=\text{odd values}}^{N_2-1} f(a+kh_2). \tag{21}$$

The factor $1/2$ in front of $I_1(a,b)$, appears because $h_2 = h_1/2$. A possible algorithm is then:

1. Choose a low number of steps to evaluate the integral, I_0, the first time, e.g. $N_0 = 10$
2. Double the number of steps, $N_1 = 2N_0$
3. Calculate the missing values by summing over the odd number of steps $\sum_{k=\text{odd values}}^{N_1-1} f(a+kh_1)$
4. Check if $E_1(a,b) = \frac{1}{3}(I_1 - I_0)$ is lower than a specific tolerance

5. If yes quit, if not, return to 2, and continue until $E_i(a,b) = \frac{1}{3}(I_{i+1} - I_i)$ is lower than the tolerance

Below is a Python implementation:

```
import numpy as np
# Function to be integrated
def f(x):
    return np.sin(x)
# step size is chosen automatically to reach the specified tolerance
def int_adaptive_trapez(lower_limit, upper_limit,func,tol):
    N0      = 10
    h       = (upper_limit-lower_limit)/N0
    area    = func(lower_limit)+func(upper_limit)
    area   *= 0.5
    val     = lower_limit
    for k in range(1,N0): # loop over k=1,..,N-1
        val   += h # midpoint value
        area  += func(val)
    area   *=h
    calc_tol = 2*tol + 1 # just larger than tol to enter the while loop
    while(calc_tol>tol):
        N = N0*2
        h = (upper_limit-lower_limit)/N
        odd_terms=0
        for k in range (1,N,2): # 1, 3, 5, ... , N-1
            val  = lower_limit + k*h
            odd_terms += func(val)
        new_area = 0.5*area + h*odd_terms
        calc_tol = abs(new_area-area)/3
        area     = new_area # store new values for next iteration
        N0       = N        # update number of slices
    print('Number of intervals = ', N)
    return area #while loop ended and we can return the area

prec=1e-8
a=0
b=np.pi
Area = int_adaptive_trapez(a,b,f,prec)
print('Numerical value = ', Area)
print('Error           = ', (2-Area)) # Analytical result is 2
```

If you compare the number of terms used in the adaptive trapezoidal rule, which was developed by halving the step size, and the adaptive midpoint rule that was derived on the basis of the theoretical error term, you will find the adaptive midpoint rule is more efficient. So why go through all this trouble? In the next section we will see that the development we did for the adaptive trapezoidal rule is closely related to Romberg integration, which is *much* more effective.

2 Romberg Integration

The adaptive algorithm for the trapezoidal rule in the previous section can be easily improved by remembering that the true integral was given by[1] :

[1] Note that all odd powers of h is equal to zero, thus the corrections are always in even powers.

$I = I_i + ch_i^2 + \mathcal{O}(h^4)$. The error term was in the previous example only used to check if the desired tolerance was achieved, but we could also have added it to our estimate of the integral to reach an accuracy to fourth order:

$$I = I_{i+1} + ch^2 + \mathcal{O}(h^4) = I_{i+1} + \frac{1}{3}\left[I_{i+1} - I_i\right] + \mathcal{O}(h^4). \tag{22}$$

As before the error term $\mathcal{O}(h^4)$, can be written as: ch^4. Now we can proceed as in the previous section: First we estimate the integral by one step size $I_i = I + ch_i^4$, next we half the step size $I_{i+1} = I + ch_{i+1}^4$ and use these two estimates to calculate the error term:

$$\begin{aligned} I_{i+1} - I_i = I - ch_{i+1}^4 - (I - ch_i^4) = -ch_{i+1}^4 + c(2h_{i+1})^4 = 15ch_{i+1}^4, \\ ch_{i+1}^4 = \frac{1}{15}\left[I_{i+1} - I_i\right] + \mathcal{O}(h^6). \end{aligned} \tag{23}$$

but now we are in the exact situation as before, we have not only the error term but the correction up to order h^4 for this integral:

$$I = I_{i+1} + \frac{1}{15}\left[I_{i+1} - I_i\right] + \mathcal{O}(h^6). \tag{24}$$

Each time we half the step size we also gain a higher order accuracy in our numerical algorithm. Thus, there are two iterations going on at the same time; one is the iteration that half the step size (i), and the other one is the increasing number of higher order terms added (which we will denote m). We need to improve our notation, and replace the approximation of the integral (I_i) with $R_{i,m}$. Equation (24), can now be written:

$$I = R_{i+1,2} + \frac{1}{15}\left[R_{i+1,2} - R_{i,2}\right] + \mathcal{O}(h^6). \tag{25}$$

A general formula valid for any m can be found by realizing:

$$I = R_{i+1,m+1} + c_m h_i^{2m+2} + \mathcal{O}(h_i^{2m+4}) \tag{26}$$

$$\begin{aligned} I &= R_{i,m+1} + c_m h_{i-1}^{2m+2} + \mathcal{O}(h_{i-1}^{2m+4}) \\ &= R_{i,m+1} + 2^{2m+2} c_m h_i^{2m+2} + \mathcal{O}(h_{i-1}^{2m+4}), \end{aligned} \tag{27}$$

where, as before $h_{i-1} = 2h_i$. Subtracting equation (26) and (27), we find an expression for the error term:

$$c_m h_i^{2m+2} = \frac{1}{4^{m+1} - 1}(R_{i,m} - R_{i-1,m}) \tag{28}$$

Then the estimate for the integral in equation (27) is:

$$I = R_{i,m+1} + \mathcal{O}(h_i^{2m+2}) \tag{29}$$

$$R_{i,m+1} = R_{i,m} + \frac{1}{4^{m+1} - 1}(R_{i+1,m} - R_{i,m}). \tag{30}$$

A possible algorithm is then:

1. Evaluate $R_{0,0} = \frac{1}{2}\left[f(a) + f(b)\right](b-a)$ as the first estimate
2. Double the number of steps, $N_{i+1} = 2N_i$ or half the step size $h_{i+1} = h_i/2$
3. Calculate the missing values by summing over the odd number of steps $\sum_{k=\text{odd values}}^{N_1-1} f(a + kh_{i+1})$
4. Correct the estimate by adding *all* the higher order error term $R_{i,m+1} = R_{i,m} + \frac{1}{4^m-1}(R_{i+1,m+1} - R_{i,m+1})$
5. Check if the error term is lower than a specific tolerance $E_{i,m}(a,b) = \frac{1}{4^{m+1}-1}(R_{i,m} - R_{i-1,m})$, if yes quit, if no goto 2, increase i and m by one

The algorithm is illustrated in figure 3.

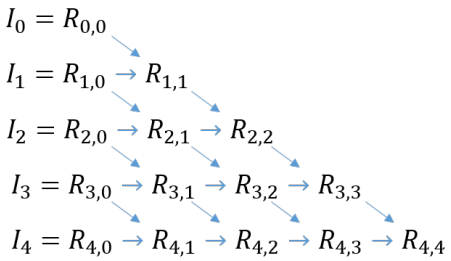

Figure 3: Illustration of the Romberg algorithm. Note that for each new evaluation of the integral $R_{i,0}$, all the correction terms $R_{i,m}$ (for $m > 0$) must be evaluated again.

Note that the tolerance term is not the correct one as it uses the error estimate for the current step, which we also use correct the integral in the current step to reach a higher accuracy. Thus the error on the integral will always be lower than the user specified tolerance. Below is a Python implementation:

```
def int_romberg(func,lower_limit, upper_limit,tol,show=False):
    """ calculates the area of func over the domain lower_limit
        to upper limit for the given tol, if show=True the triangular
        array of intermediate results are printed """
    Nmax = 100
    R = np.empty([Nmax,Nmax]) # storage buffer
    h = (upper_limit-lower_limit) # step size
    R[0,0]    =.5*(func(lower_limit)+func(upper_limit))*h
    N = 1
    for i in range(1,Nmax):
        h /= 2
        N *= 2
        odd_terms=0
        for k in range (1,N,2): # 1, 3, 5, ... , N-1
            val        = lower_limit + k*h
            odd_terms += func(val)
                # add the odd terms to the previous estimate
        R[i,0]   = 0.5*R[i-1,0] + h*odd_terms
        for m in range(0,i): # m = 0, 1, ..., i-1
                        # add all higher order terms in h
```

```
            R[i,m+1]   = R[i,m] + (R[i,m]-R[i-1,m])/(4**(m+1)-1)
                # check tolerance, best guess
        calc_tol = abs(R[i,i]-R[i-1,i-1])
        if(calc_tol<tol):
            break  # estimated precision reached
    if(i == Nmax-1):
        print('Romberg routine did not converge after ',
              Nmax, 'iterations!')
    else:
        print('Number of intervals = ', N)

    if(show==True):
        elem = [2**idx for idx in range(i+1)]
        print("Steps StepSize Results")
        for idx in range(i+1):
            print(elem[idx],' ',
                  "{:.6f}".format((upper_limit-lower_limit)/2**idx),end = ' ')
            for l in range(idx+1):
                print("{:.6f}".format(R[idx,l]),end = ' ')
            print('')
    return R[i,i] #return the best estimate
```

Note that the Romberg integration only uses 32 function evaluations to reach a precision of 10^{-8}, whereas the adaptive midpoint and trapezoidal rule in the previous section uses 20480 and 9069 function evaluations, respectively.

2.1 Gaussian Quadrature

Many of the methods we have looked into are of the type:

$$\int_a^b f(x)dx = \sum_{k=0}^{N-1} \omega_k f(x_k), \tag{31}$$

where the function is evaluated at fixed interval. For the midpoint rule $\omega_k = h$ for all values of k, for the trapezoid rule $\omega_k = h/2$ for the endpoints and h for all the interior points. For the Simpsons rule (see exercise) $\omega_k = h/3, 4h/3, 2h/3, 4h/3, \ldots, 4h/3, h/3$. Note that all the methods we have looked at so far samples the function in equal spaced points, $f(a + kh)$, for $k = 0, 1, 2 \ldots, N - 1$. If we now allow for the function to be evaluated at unevenly spaced points, we can do a lot better. This realization is the basis for Gaussian Quadrature. We will explore this in the following, but to make the development easier and less cumbersome, we transform the integral from the domain $[a, b]$ to $[-1, 1]$:

$$\int_a^b f(t)dt = \frac{b-a}{2} \int_{-1}^{1} f(x)dx \text{ , where:} \tag{32}$$

$$x = \frac{2}{b-a}t - \frac{b+a}{b-a}. \tag{33}$$

The factor in front comes from the fact that $dt = (b - a)dx/2$, thus we can develop our algorithms on the domain $[-1, 1]$, and then do the transformation back using: $t = (b - a)x/2 + (b + a)/2$.

Notice.

The idea we will explore is as follows: If we can approximate the function to be integrated on the domain $[-1, 1]$ (or on $[a, b]$) as a polynomial of as *large a degree as possible*, then the numerical integral of this polynomial will be very close to the integral of the function we are seeking.

This idea is best understood by a couple of examples. Assume that we want to use $N = 1$ in equation (31):

$$\int_{-1}^{1} f(x)\,dx \simeq \omega_0 f(x_0). \tag{34}$$

We now choose $f(x)$ to be a polynomial of as large a degree as possible, but with the requirement that the integral is exact. If $f(x) = 1$, we get:

$$\int_{-1}^{1} f(x)\,dx = \int_{-1}^{1} 1\,dx = 2 = \omega_0, \tag{35}$$

hence $\omega_0 = 2$. If we choose $f(x) = x$, we get:

$$\int_{-1}^{1} f(x)\,dx = \int_{-1}^{1} x\,dx = 0 = \omega_0 f(x_0) = 2x_0, \tag{36}$$

hence $x_0 = 0$.

The Gaussian integration rule for $N = 1$ is:

$$\int_{-1}^{1} f(x)\,dx \simeq 2f(0), \text{ or:}$$
$$\int_{a}^{b} f(t)\,dt \simeq \frac{b-a}{2}\,2f(\frac{b+a}{2}) = (b-a)f(\frac{b+a}{2}). \tag{37}$$

This equation is equal to the midpoint rule, by choosing $b = a + h$ we reproduce equation (1). If we choose $N = 2$:

$$\int_{-1}^{1} f(x)\,dx \simeq \omega_0 f(x_0) + \omega_1 f(x_1), \tag{38}$$

we can show that now $f(x) = 1$, x, x^2 x^3 can be integrated exact:

$$\int_{-1}^{1} 1\,dx = 2 = \omega_0 f(x_0) + \omega_1 f(x_1) = \omega_0 + \omega_1\,, \tag{39}$$

$$\int_{-1}^{1} x\,dx = 0 = \omega_0 f(x_0) + \omega_1 f(x_1) = \omega_0 x_0 + \omega_1 x_1\,, \tag{40}$$

$$\int_{-1}^{1} x^2\,dx = \frac{2}{3} = \omega_0 f(x_0) + \omega_1 f(x_1) = \omega_0 x_0^2 + \omega_1 x_1^2\,, \tag{41}$$

$$\int_{-1}^{1} x^3\,dx = 0 = \omega_0 f(x_0) + \omega_1 f(x_1) = \omega_0 x_0^3 + \omega_1 x_1^3\,, \tag{42}$$

hence there are four unknowns and four equations. The solution is: $\omega_0 = \omega_1 = 1$ and $x_0 = -x_1 = 1/\sqrt{3}$.

The Gaussian integration rule for $N = 2$ is:

$$\int_{-1}^{1} f(x)\,dx \simeq f(-\frac{1}{\sqrt{3}}) + f(\frac{1}{\sqrt{3}})\,, \text{ or:} \tag{43}$$

$$\int_{a}^{b} f(x)\,dx \simeq \frac{b-a}{2}\left[f(-\frac{b-a}{2}\frac{1}{\sqrt{3}} + \frac{b+a}{2}) + f(\frac{b-a}{2}\frac{1}{\sqrt{3}} + \frac{b+a}{2})\right]. \tag{44}$$

```
def int_gaussquad2(func, lower_limit, upper_limit):
    x   = np.array([-1/np.sqrt(3.),1/np.sqrt(3)])
    w   = np.array([1, 1])
    xp  = 0.5*(upper_limit-lower_limit)*x
    xp += 0.5*(upper_limit+lower_limit)
    area = np.sum(w*func(xp))
    return area*0.5*(upper_limit-lower_limit)
```

The case N=3. For the case $N = 3$, we find that $f(x) = 1, x, x^2, x^3, x^4, x^5$ can be integrated exactly:

$$\int_{-1}^{1} 1\, dx = 2 = \omega_0 + \omega_1 + \omega_2\,, \tag{45}$$

$$\int_{-1}^{1} x\, dx = 0 = \omega_0 x_0 + \omega_1 x_1 + \omega_2 x_2\,, \tag{46}$$

$$\int_{-1}^{1} x^2\, dx = \frac{2}{3} = \omega_0 x_0^2 + \omega_1 x_1^2 + \omega_2 x_2^2\,, \tag{47}$$

$$\int_{-1}^{1} x^3\, dx = 0 = \omega_0 x_0^3 + \omega_1 x_1^3 + \omega_2 x_2^3\,, \tag{48}$$

$$\int_{-1}^{1} x^4\, dx = \frac{2}{5} = \omega_0 x_0^4 + \omega_1 x_1^4 + \omega_2 x_2^4\,, \tag{49}$$

$$\int_{-1}^{1} x^5\, dx = 0 = \omega_0 x_0^5 + \omega_1 x_1^5 + \omega_2 x_2^5\,, \tag{50}$$

the solution to these equations are $\omega_{0,1,2} = 5/9, 8/9, 5/9$ and $x_{1,2,3} = -\sqrt{3/5}, 0, \sqrt{3/5}$. Below is a Python implementation:

```
def int_gaussquad3(lower_limit, upper_limit,func):
    x  = np.array([-np.sqrt(3./5.),0.,np.sqrt(3./5.)])
    w  = np.array([5./9., 8./9., 5./9.])
    xp = 0.5*(upper_limit-lower_limit)*x
    xp += 0.5*(upper_limit+lower_limit)
    area = np.sum(w*func(xp))
    return area*0.5*(upper_limit-lower_limit)
```

Note that the Gaussian quadrature converges very fast. From $N = 2$ to $N = 3$ function evaluation we reduce the error (in this specific case) from 6.5% to 0.1%. Our standard trapezoidal formula needs more than 20 function evaluations to achieve this, the Romberg method uses 4-5 function evaluations. How can this be? If we use the standard Taylor formula for the function to be integrated, we know that for $N = 2$ the Taylor formula must be integrated up to x^3, so the error term is proportional to $h^4 f^{(4)}(\xi)$ (where ξ is some x-value in $[a, b]$). h is the step size, and we can replace it with $h \sim (b - a)/N$, thus the error scale as c_N/N^4 (where c_N is a constant). Following the same argument, we find for $N = 3$ that the error term is $h^6 f^{(6)}(\xi)$ or that the error term scale as c_N/N^6. Each time we increase N by a factor of one, the error term reduces by N^2. Thus if we evaluate the integral for $N = 10$, increasing to $N = 11$ will reduce the error by a factor of $11^2 = 121$.

2.2 Error term on Gaussian Integration

The Gaussian integration rule of order N integrates exactly a polynomial of order $2N - 1$. From Taylors error formula, we can easily see that the error term must be of order $2N$, and be proportional to $f^{(2N)}(\eta)$, see [1] for more details

on the derivation of error terms. The drawback with an analytical error term derived from series expansion is that it involves the derivative of the function. As we have already explained, this is very unpractical and it is much more practical to use the methods described in section 1.4. Let us consider this in more detail, assume that we evaluate the integral using first a Gaussian integration rule with N points, and then $N+1$ points. Our estimates of the "exact" integral, I, would then be:

$$I = I_N + ch_N^{2N}, \tag{51}$$

$$I = I_{N+1} + ch_{N+1}^{2N+1}. \tag{52}$$

In principle $h_{N+1} \neq h_N$, but in the following we will assume that $h_N \simeq h_{N+1}$, and $h \ll 1$. Subtracting equation (51) and (52) we can show that a reasonable estimate for the error term ch^{2N} would be:

$$ch^N = I_{N+1} - I_N. \tag{53}$$

If this estimate is lower than a given tolerance we can be quite confident that the higher order estimate I_{N+1} approximate the true integral within our error estimate. This is the method implemented in SciPy, `integrate.quadrature`

2.3 Common Weight functions for Classical Gaussian Quadratures

3 Integrating functions over an infinite range

Integrating a function over an infinite range can be done by the following trick. Assume that we would like to evaluate

$$\int_a^\infty f(x)dx. \tag{54}$$

If we introduce the following substitution

$$z = \frac{x-a}{1+x-a}, \tag{55}$$

or equivalently

$$x = a + \frac{z}{1-z}, \tag{56}$$

then if $x = a$, $z = 0$, and if $x \to \infty$ then $z \to 1$, hence:

$$\int_a^\infty f(x)dx = \int_0^1 f(a + \frac{z}{1-z})\frac{dz}{(1-z)^2}. \tag{57}$$

3.1 Which method to use in a specific case? (NOT COMPLETED)

There are no general answers to this question, and one need to decide from case to case. If computational speed is not an issue, and the function to be integrated can be evaluated at any points all the methods above can be used. If the function to be integrated is a set of observations at different times, that might be unevenly spaced, I would use the midpoint rule:

$$I(a,b) = \int_a^b f(x)dx \simeq \sum_{k=0}^{N-1} M(x_k, x_k + h) = \sum_{k=0}^{N-1} h_i f(x_k + \frac{h_i}{2}) \qquad (58)$$

This is because we do not know anything about the function between the points, only when it is observed, and the formula uses only the information at the observation points. There is a second more subtle reason, and that is the fact that in many cases the observations a different times are the *average* value of the observable quantity and it those cases the midpoint rule would be the exact answer.

Exercise 1: Numerical Integration

a) Show that for a linear function, $y = a \cdot x + b$ both the trapezoidal rule and the rectangular rule are exact

b) Consider $I(a,b) = \int_a^b f(x)dx$ for $f(x) = x^2$. The analytical result is $I(a,b) = \frac{b^3 - a^3}{3}$. Use the Trapezoidal and Midpoint rule to evaluate these integrals and show that the error for the Trapezoidal rule is exactly twice as big as the Midpoint rule.

c) Use the fact that the error term on the trapezoidal rule is twice as big as the midpoint rule to derive Simpsons formula: $I(a,b) = \sum_{k=0}^{N-1} I(x_k, x_k + h) = \frac{h}{6}\left[f(a) + 4f(a + \frac{h}{2}) + 2f(a + h) + 4f(a + 3\frac{h}{2}) + 2f(a + 2h) + \cdots + f(b)\right]$ Hint: $I(x_k, x_k + h) = M(x_k, x_k + h) + E_M$ (midpoint rule) and $I(x_k, x_k + h) = T(x_k, x_k + h) + E_T = T(x_k, x_k + h) - 2E_M$ (trapezoidal rule).

Solution. Simpsons rule is an improvement over the midpoint and trapezoidal rule. It can be derived in different ways, we will make use of the results in the previous section. If we assume that the second derivative is reasonably well behaved on the interval x_k and $x_k + h$ and fairly constant we can assume that $f''(\eta) \simeq f''(\overline{\eta})$, hence $E_T = -2E_M$.

$$I(x_k, x_k + h) = M(x_k, x_k + h) + E_M \text{ (midpoint rule)} \qquad (59)$$

$$\begin{aligned} I(x_k, x_k + h) &= T(x_k, x_k + h) + E_T \\ &= T(x_k, x_k + h) - 2E_M \text{ (trapezoidal rule)}, \end{aligned} \qquad (60)$$

we can now cancel out the error term by multiplying the first equation with 2 and adding the equations:

$$3I(x_k, x_k + h) = 2M(x_k, x_k + h) + T(x_k, x_k + h) \tag{61}$$

$$= 2f(x_k + \frac{h}{2})h + [f(x_k + h) + f(x_k)]\,\frac{h}{2} \tag{62}$$

$$I(x_k, x_k + h) = \frac{h}{6}\left[f(x_k) + 4f(x_k + \frac{h}{2}) + f(x_k + h)\right]. \tag{63}$$

Now we can do as we did in the case of the trapezoidal rule, sum over all the elements:

$$\begin{aligned} I(a,b) &= \sum_{k=0}^{N-1} I(x_k, x_k + h) \\ &= \frac{h}{6}\Big[f(a) + 4f(a + \frac{h}{2}) + 2f(a + h) + 4f(a + 3\frac{h}{2}) \\ &\quad +2f(a + 2h) + \cdots + f(b)] \qquad (64) \\ &= \frac{h'}{3}\left[f(a) + f(b) + 4\sum_{k=\text{odd}}^{N-2} f(a + kh') + 2\sum_{k=\text{even}}^{N-2} f(a + kh')\right], \qquad (65) \end{aligned}$$

note that in the last equation we have changed the step size $h = 2h'$.

d) Show that for $N = 2$ ($f(x) = 1, x, x^3$), the points and Gaussian quadrature rule for $\int_0^1 x^{1/2} f(x)\,dx$ is $\omega_{0,1} = -\sqrt{70}150 + 1/3, \sqrt{70}150 + 1/3$ and $x_{0,1} = -2\sqrt{70}63 + 5/9, 2\sqrt{70}63 + 5/9$

1. Integrate $\int_0^1 x^{1/2} \cos x\,dx$ using the rule derived in the exercise above and compare with the standard Gaussian quadrature rule for ($N = 2$, and $N = 3$).

e) Make a Python program that uses the Midpoint rule to integrate experimental data that are unevenly spaced and given in the form of two arrays.

References

[1] Josef Stoer and Roland Bulirsch. *Introduction to Numerical Analysis*, volume 12. Springer Science & Business Media, 2013.