Modeling and Computational Engineering

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

University of Stavanger

Aug 27, 2019

(Work in Progress) The purpose of this document is to explain how computers solve mathematical models. Many of the most common numerical methods is presented, we show how to implement them in Python, and discuss the limitations. The mathemathical formalism is kept to a minimum. All the material is available at github[a]. For each of the chapter there is a Jupyter notebook[b]. This makes it possible to run all the codes in this document. We strongly recommend to install Python from Anaconda[c]. All documents have been prepared using doconce[d].

[a] `https://github.com/ahiorth/CompEngineering`

[b] `https://github.com/ahiorth/CompEngineering/tree/master/pub/chapters`

[c] `https://www.anaconda.com/`

[d] `https://github.com/ahiorth/CompEngineering/tree/master/pub/chapters`

Preface

What does computers do better than humans? What is it possible to compute? These questions have not been fully answered yet, and in the coming years we will most likely see that the boundaries for what computers can do will expand significantly. Many of the fundamental laws in nature have been known for quite some time, but still it is almost impossible to predict the behavior of water (H_2O) from quantum mechanics. The most sophisticated super computers runs for days and are only able to simulate the behavior of molecules in a couple of seconds, almost too short to extract meaningful thermodynamic properties. This leads to another interesting question: What does humans do better than machines? A large part of the answer to this question is *modeling*. Modeling is the ability to break a complicated, unstructured problem into smaller pieces that can be solved by computers or by other means. Modeling requires *domain knowledge*, one need to understand the system well enough to make the correct or the most efficient simplifications. The process usually starts with some experimental data that one would like to understand, it could be the increasing temperature in the atmosphere or sea, it could be changes in the chemical composition of a fluid passing through a rock. The modeler then makes a mental image, which includes a set of mechanisms that could be the cause of the observed data. These mechanisms then needs to be formulated mathematically. How can we know if a model of a system is good? First of all, a good model is a model that do not break any of the fundamental laws of nature, such as mass (assuming non relativistic effects) and energy conservation. Even if you are searching for new laws of nature, you have to make sure that your model respect the existing laws, because then a deviation from your model and the observations could be a hint of the new physics you are searching for. Secondly, the model must be able to match the observable data, with a limited set of variables. The variables should be determined from data, and then the model should be able to make some predictions that can be tested. Thus, the true purpose of the model is not only to match experimental data, but serve as a framework where the underlying mechanisms of the process can be understood. This is done by making model predictions, test them, and improve the model.

In this course our main focus will be on how to use computers to solve models. We will show you through exercises how a mathematical model of a physical system can be made, and you will have the possibility to explore the model. Computers are extremely useful, they can solve problems that would be impossible to solve by hand. However, it is extremely important to know about the limitations and strength of various algorithms. One need to have a toolbox of various algorithms that can be employed depending on the problem one are studying. Sometimes speed is not an issue, and one can use simpler algorithms, but in many cases *speed is an issue*. Thus it is important to not waste computational time when it is not needed, we will encounter examples of this many times in this course. Why should you spend time learning about algorithms that have been implemented already in a software that most likely can be downloaded for free? There are many answers to this question, some more practical and some that goes deeper. Lets start with the practical considerations: Often you encounter a problem that needs to be solved by a computer, it could be as simple as to integrate some production data in a spreadsheet to calculate the total production, or it could be to fit a function with more than one variable to some data. Once you have this problem, and starting to ask Mr. Google for a solution, you will quickly realize that there are numerous ways of achieving what you want. By educating yourself within the most basic numerical methods, presented in this course, you will be able to judge for yourself which method to use in a specific case. Another motivation is that development of most of the different numerical methods are *not that difficult*, they usually follow a very similar pattern, but there are some ”tricks”. It is extremely useful to learn these tricks, they can be adopted to a range of different problems, many are easily implemented in a spreadsheet. There are some more deeper arguments, and that is that the numerical methods are developed to solve a *general* problem. Most of the time we work with *specific* problems, and we would like to have an algorithm that is optimal for our problem that goes beyond only choosing the right one. Having understood and learned all the cool tricks that was used in the development of the algorithm in the general case, is a starting point for adopting the algorithm to your specific situation. Secondly development of an algorithm is a concrete case of *Computational Thinking*. Computational thinking is not necessarily related to computers and programming, but it is a way of structuring your work into precise statements that are being executed one at a time in a specific order. By learning about algorithmic development, you will train yourself in the art of computational thinking, which is a useful skill in all kind of problem solving.

November 2018 *Aksel Hiorth*

Contents

Preface vii

1 Finite differences 1

1.1 Numerical derivatives 1

1.2 Taylor Polynomial Approximation 3
1.2.1 Evaluation of polynomials 5

1.3 Calculating Derivatives of Functions 6
1.3.1 Big $\mathcal{O}$ notation 8
1.3.2 Round off Errors 8

1.4 Higher Order Derivatives 9

2 Linear and nonlinear equations 13

2.1 Solving linear equations 14
2.1.1 Gauss-Jordan elimination 15
2.1.2 Pivoting 17
2.1.3 LU decomposition 18

2.2 Example: Linear regression 19

2.3 Example: Solving the heat equation regression 20

2.4 Singular Value Decomposition 20

2.5 QR factorization 20

2.6 Solving nonlinear equations 20

3 Numerical integration 21

3.1 Numerical Integration 21
3.1.1 The Midpoint Rule 22
3.1.2 The Trapezoidal Rule 23
3.1.3 Numerical Errors on Integrals 25
3.1.4 Practical Estimation of Errors on Integrals 28

3.2 Romberg Integration 30
3.2.1 Gaussian Quadrature 33
3.2.2 Error term on Gaussian Integration 37
3.2.3 Common Weight functions for Classical Gaussian Quadratures 37
3.2.4 Which method to use in a specific case? 37

4 Ordinary differential equations 41

5 Monte Carlo Methods 43

References 45

List of Exercises

Exercise 3.1: Numerical Integration 38

Finite differences 1

The mathematics introduced in this chapter is absolutely essential in order to understand the development of numerical algorithms. We strongly advice you to study it carefully, implement python scripts and investigate the results, reproduce the analytical derivations and compare with the numerical solutions.

1.1 Numerical derivatives

The solution to a physical model is usually a function. The function could describe the temperature evolution of the earth, it could be growth of cancer cells, the water pressure in an oil reservoir, the list is endless. If we can solve the model analytically, the answer is given in terms of a continuous function. Most of the models cannot be solved analytically, then we have to rely on computers to help us. The computer does not have any concept of continuous functions, a function is always evaluated at some point in space and/or time. Assume for simplicity that the solution to our problem is $f(x) = \sin x$, and we would like to visualize the solution. How many points do we need in our plot to approximate the true function? In figure 1.1, there is a plot of $\sin x$ on the interval $[-\pi, \pi]$.

From the figure we see that in some areas only a couple of points are needed in order to represent the function well, and in some areas more points are needed. To state it more clearly; between $[-1, 1]$ a linear

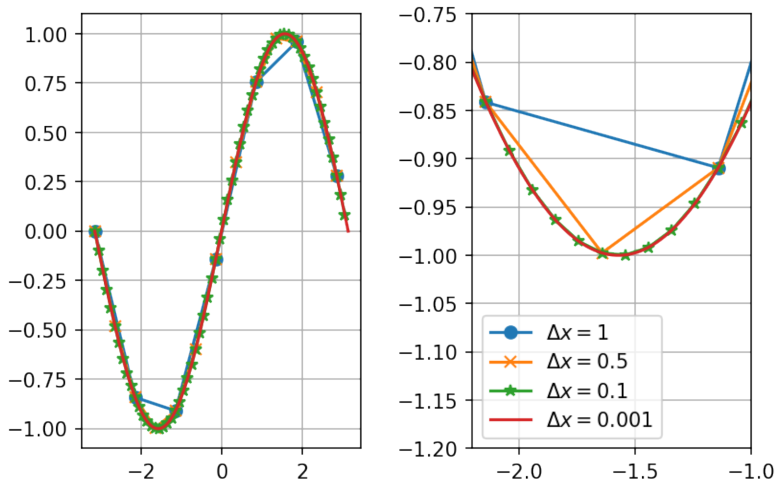

Fig. 1.1 A plot of $\sin x$ for different spacing of the x-values.

function (few points) approximate $\sin x$ well, whereas in the area where the derivative of the function changes e.g. in $[-2, -1]$, we need the points to be more closely spaced to capture the behavior of the true function.

Discretization

To represent a function of space and/or time in a computer, the function needs to be discretized. When a function is discretized it leads to discretization errors.

Why do we care about the number of points? In many cases the function we would like to evaluate can take a very long time to evaluate. Sometimes simulation time is not an issue, then we can use a large number of function evaluations. However, in many applications simulation time *is an issue*, and it would be good to know where the points needs to be closely spaced, and where we can manage with only a few points.

What is a *good representation* representation of the true function? We cannot rely on visual inspection. In the next section we will show how Taylor polynomial representation of a function is a natural starting point to answer this question.

1.2 Taylor Polynomial Approximation

There are many ways of representing a function, but perhaps one of the most widely used is Taylor polynomials. Taylor series are the basis for solving ordinary and differential equations, simply because it makes it possible to evaluate any function with a set of limited operations: *addition, subtraction, and multiplication.* The Taylor polynomial, $P_n(x)$ of degree n of a function $f(x)$ at the point c is defined as:

Taylor polynomial:

$$\begin{aligned} P_n(x) &= f(c) + f'(c)(x-c) + \frac{f''(c)}{2!}(x-c)^2 + \cdots + \frac{f^{(n)}(c)}{n!}(x-c)^n \\ &= \sum_{k=0}^{n} \frac{f^{(n)}(c)}{k!}(x-c)^k. \end{aligned} \tag{1.1}$$

If the series is around the point $c = 0$, the Taylor polynomial $P_n(x)$ is often called a Maclaurin polynomial, more examples can be found here[1]. If the series converge (i.e. that the higher order terms approach zero), then we can represent the function $f(x)$ with its corresponding Taylor series around the point $x = c$:

$$f(x) = f(c) + f'(c)(x-c) + \frac{f''(c)}{2!}(x-c)^2 + \cdots = \sum_{k=0}^{\infty} \frac{f^{(n)}}{k!}(x-c)^k. \tag{1.2}$$

The Maclaurin series of $\sin x$ is:

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots = \sum_{k=0}^{\infty} \frac{(-1)^n}{(2n+1)!} x^{2n+1}. \tag{1.3}$$

In figure 1.2, we show the first nine terms in the Maclaurin series for $\sin x$ (all even terms are zero).

Note that we get a decent representation of $\sin x$ on the domain, by *only knowing the function and its derivative in a single point.* The error term in Taylors formula, when we represent a function with a finite number of polynomial elements is given by:

[1] `https://en.wikipedia.org/wiki/Taylor_series`

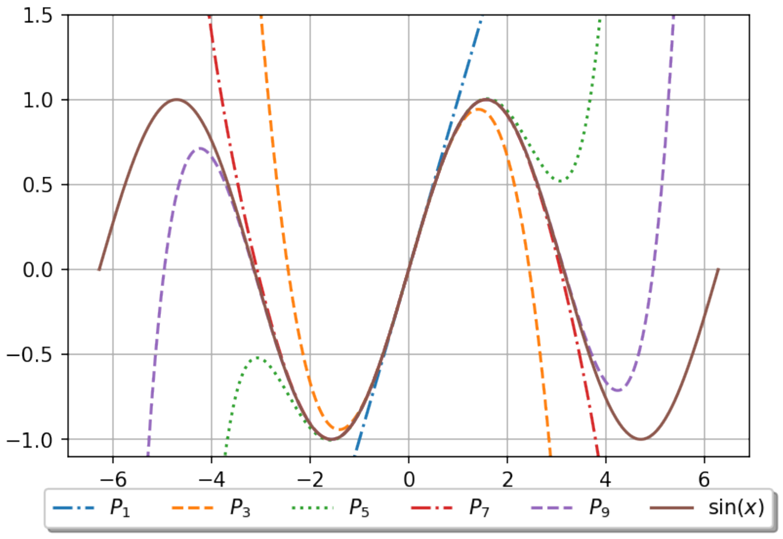

Fig. 1.2 Up to ninth order in the Maclaurin series of $\sin x$.

Error term in Taylors formula:

$$R_n(x) = f(x) - P_n(x) = \frac{f^{(n+1)}(\eta)}{(n+1)!}(x-c)^{n+1}$$
$$= \frac{1}{n!}\int_c^x (x-\tau)^n f^{(n+1)}(\tau)d\tau, \tag{1.4}$$

for some η in the domain $[x, c]$.

If we want to calculate $\sin x$ to a precision lower than a specified value we can do it as follows:

```
import numpy as np

# Sinus implementation using the Maclaurin Serie
# By setting a value for eps this value will be used
# if not provided
def my_sin(x,eps=1e-16):
    f = power = x
    x2 = x*x
    sign = 1
    i=0
    while(power>=eps):
        sign = - sign
        power *= x2/(2*i+2)/(2*i+3)
        f += sign*power
        i += 1
    print('No function evaluations: ', i)
    return f

x=0.8
eps = 1e-9
print(my_sin(x,eps), 'error = ', np.sin(x)-my_sin(x,eps))
```

This implementation needs some explanation:

- The error term is given in equation (1.4), and it is a even power in x. We do not which η to use in equation (1.4), thus we use a trick and simply say that the error term is smaller than the highest order term. Thus, we stop the evaluation if the highest order term in the series is lower than the uncertainty. Thus, in practice we add the error term to the function evaluation, our estimate will always be better than the specified accuracy.
- We evaluate the polynomials in the Taylor series by using the previous values too avoid too many multiplications within the loop, we do this by using the following identity:

$$\begin{aligned}
\sin x &= \sum_{k=0}^{\infty}(-1)^n t_n, \text{ where: } t_n \equiv \frac{x^{2n+1}}{(2n+1)!}, \text{ hence :}\\
t_{n+1} &= \frac{x^{2(n+1)+1}}{(2(n+1)+1)!} = \frac{x^{2n+1}x^2}{(2n+1)!(2n+2)(2n+3)}\\
&= t_n\frac{x^2}{(2n+2)(2n+3)} \qquad (1.5)
\end{aligned}$$

1.2.1 Evaluation of polynomials

How to evaluate a polynomial of the type: $p_n(x) = a_0 + a_1x + a_2x^2 + \cdots + a_nx^n$? We already saw a hint in the previous section that it can be done in different ways. One way is simply to do:

```
pol = a[0]
for i in range(1,n+1):
    pol = pol + a[i]*x**i
```

Note that there are n additions, whereas there are $1 + 2 + 3 + \cdots + n = n(n+1)/2$ multiplications for all the iterations. Instead of evaluating the powers all over in each loop, we can use the previous calculation to save the number of multiplications:

```
pol = a[0] + a[1]*x
power = x
for i in range(2,n+1):
    power  = power*x
    pol    = pol + a[i]*power
```

In this case there are still n additions, but now there are $2n-1$ multiplications. For $n = 15$, this amounts to 120 for the first, and 29 for the second method. Polynomials can also be evaluated using *nested multiplication*:

$$\begin{aligned}
p_1 &= a_0 + a_1 x \\
p_2 &= a_0 + a_1 x + a_2 x^2 = a_0 + x(a_1 + a_2 x) \\
p_3 &= a_0 + a_1 x + a_2 x^2 + a_3 x^3 = a_0 + x(a_1 + x(a_2 + a_3 x)) \\
&\vdots
\end{aligned} \tag{1.6}$$

and so on. This can be implemented as:

```
pol = a[n]
for i in range(n-1,1,-1):
    pol  = a[i] + pol*x
```

In this case we only have n multiplications. So if you know beforehand exactly how many terms is needed to calculate the series, this method would be the preferred method, and is implemented in NumPy as `polyval`[2].

1.3 Calculating Derivatives of Functions

indexforward difference

The derivative of a function can be calculated using the definition from calculus:

$$f'(x) = \lim_{h\to 0} \frac{f(x+h) - f(x)}{h} \simeq \frac{f(x+h) - f(x)}{h}. \tag{1.7}$$

Not that h can be both positive and negative, if h is positive equation (1.7) is termed *forward difference*, because we use the function value on the right ($f(x + |h|)$). If on the other hand h is negative equation (1.7) is termed *backward difference*, because we use the value to the left

[2] `https://docs.scipy.org/doc/numpy/reference/generated/numpy.polyval.html#r138ee7027ddf-1`

$(f(x-|h|))$. ($|h|$ is the absolute value of h). In the computer we cannot take the limit, $h \to 0$, a natural question is then: What value to use for h? In figure 1.3, we have evaluated the numerical derivative of $\sin x$, using the formula in equation (1.7) for different step sizes h.

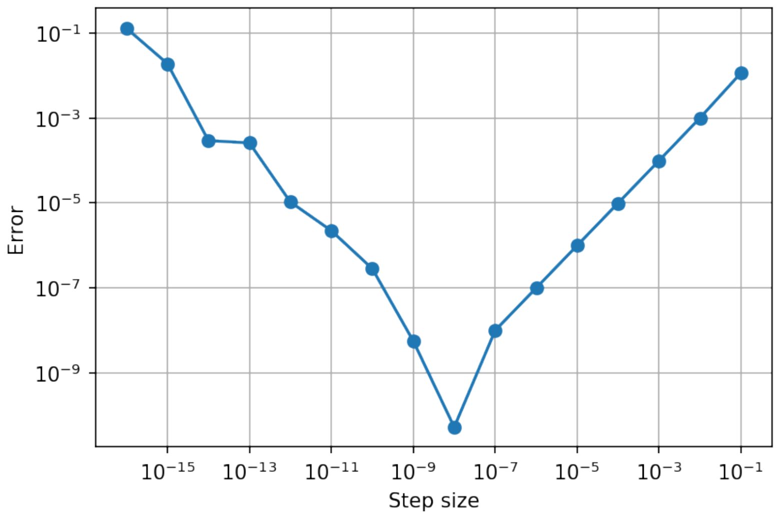

Fig. 1.3 Error in the numerical derivative of $\sin x$ at $x = 0.2$ for different step size.

We clearly see that the error depends on the step size, but there is a minimum; choosing a step size too large give a poor estimate and choosing a too low step size give an even worse estimate. The explanation for this behavior is two competing effects: *mathematical approximation* and *round off errors*. Let us consider approximation or truncation error first. By using the Taylor expansion in equation (1.2) and expand about x and the error formula (1.4), we get:

$$f(x+h) = f(x) + f'(x)h + \frac{h^2}{2}f''(\eta) \text{ , hence:}$$
$$f'(x) = \frac{f(x+h)-f(x)}{h} - \frac{h}{2}f''(\eta), \tag{1.8}$$

for some η in $[x, x+h]$. Thus the error to our approximation is $hf''(\eta)/2$, if we reduce the step size by a factor of 10 the error is reduced by a factor of 10. Inspecting the graph, we clearly see that this is correct as the step size decreases from 10^{-1} to 10^{-8}. When the step size decreases more, there is an increase in the error. This is due to round off errors, and can be understood by looking into how numbers are stored in a computer.

1.3.1 Big $\mathcal{O}$ notation

example[3]

1.3.2 Round off Errors

In a computer a floating point number,x, is represented as:

$$x = \pm q 2^m. \tag{1.9}$$

Most computers are 64-bits, then one bit is reserved for the sign, 52 for the fraction (q) and 11 for the exponent (m) (for a graphic illustration see Wikipedia[4]). what is the largest *floating point* number the computer can represent? Since m contains 11 bits, m can have the maximal value $m = 2^{11} = 1024$, and then the largest value is close to $2^{1024} \simeq 10^{308}$. If you enter `print(10.1*10**(308))` in Python the answer will be `Inf`. If you enter `print(10*10**(308))`, Python will give an answer. This is because the number $10.1 \cdot 10^{308}$ is floating point number, whereas 10^{309} is an *integer*, and Python does something clever when it comes to representing integers. Python has a third numeric type called long int, which can use the available memory to represent an integer.

10^{308} is the largest number, but what is the highest precision we can use, or how many decimal places can we use for a floating point number? Since there are 52 bits for the fraction, there are $1/2^{52} \simeq 10^{-16}$ decimal places. As an example the value of π is $3.14159265358979323846264338\ldots$, but in Python it can only be represented by 16 digits: 3.141592653589793. In principle it does not sound so bad to have an answer accurate to 16 digits, and it is much better than most experimental results. So what is the problem? One problem that you should be aware of is that round off errors can be a serious problem when we subtract two numbers that are very close to one another. If we implement the following program in Python:

```
h=1e-16
x = 2.1 + h
```

[3] `https://rob-bell.net/2009/06/a-beginners-guide-to-big-o-notation/`
[4] `https://en.wikipedia.org/wiki/Double-precision_floating-point_format`

```
y = 2.1 - h
print((x-y)/h)
```

we expect to get the answer 2, but instead we get zero. By changing h to a higher value, the answer will get closer to 2.

Armed with this knowledge of round off errors, we can continue to analyze the result in figure 1.3. The round off error when we represent a floating point number x in the machine will be of the order $x/10^{16}$ (*not* 10^{-16}). In general, when we evaluate a function the error will be of the order $\epsilon|f(x)|$, where $\epsilon \sim 10^{-16}$. Thus equation (1.8) is modified in the following way when we take into account the round off errors:

$$f'(x) = \frac{f(x+h) - f(x)}{h} \pm \frac{2\epsilon|f(x)|}{h} - \frac{h}{2}f''(\eta), \tag{1.10}$$

we do not know the sign of the round off error, so the total error R_2 is:

$$R_2 = \frac{2\epsilon|f(x)|}{h} + \frac{h}{2}|f''(\eta)|. \tag{1.11}$$

We have put absolute values around the function and its derivative to get the maximal error, it might be the case that the round off error cancel part of the truncation error. However, the round off error is random in nature and will change from machine to machine, and each time we run the program. Note that the round off error increases when h decreases, and the approximation error decreases when h decreases. This is exactly what we see in the figure above. We can find the best step size, by differentiating R_2 and put it equal to zero:

$$\frac{dR_2}{dh} = -\frac{2\epsilon|f(x)|}{h^2} + \frac{1}{2}f''(\eta) = 0$$
$$h = 2\sqrt{\epsilon\left|\frac{f(x)}{f''(\eta)}\right|} \simeq 2 \cdot 10^{-8}, \tag{1.12}$$

In the last equation we have assumed that $f(x)$ and its derivative is 1. This step size corresponds to an error of order $R_2 \sim 10^{-8}$. Inspecting the result in figure 1.3. we see that the minimum is located at $h \sim 10^{-8}$.

1.4 Higher Order Derivatives

There are more ways to calculate the derivative of a function, than the formula given in equation (1.8). Different formulas can be derived by using Taylors formula in (1.2), usually one expands about $x \pm h$:

$$\begin{aligned} f(x+h) &= f(x) + f'(x)h + \frac{h^2}{2}f''(x) + \frac{h^3}{3!}f^{(3)}(x) + \frac{h^4}{4!}f^{(4)}(x) + \cdots \\ f(x-h) &= f(x) - f'(x)h + \frac{h^2}{2}f''(x) - \frac{h^3}{3!}f^{(3)}(x) + \frac{h^4}{4!}f^{(3)}(x) - \cdots . \end{aligned} \tag{1.13}$$

If we add these two equations, we get an expression for the second derivative, because the first derivative cancels out. But we also observe that if we subtract these two equations we get an expression for the first derivative that is accurate to a higher order than the formula in equation (1.7), hence:

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} - \frac{h^2}{6}f^{(3)}(\eta), \tag{1.14}$$

$$f''(x) = \frac{f(x+h) + f(x-h) - 2f(x)}{h^2} + \frac{h^2}{12}f^{(4)}(\eta), \tag{1.15}$$

for some η in $[x, x+h]$. In figure 1.4, we have plotted equation (1.8), (1.14), and (1.15) for different step sizes. The derivative in equation (1.14), gives a higher accuracy than equation (1.8) for a larger step size, as can bee seen in figure 1.4.

We can perform a similar error analysis as we did before, and then we find for equation (1.14) and (1.15) that the total numerical error is:

$$R_3 = \frac{\epsilon|f(x)|}{h} + \frac{h^2}{6}f^{(3)}(\eta), \tag{1.16}$$

$$R_4 = \frac{4\epsilon|f(x)|}{h^2} + \frac{h^2}{12}f^{(4)}(\eta), \tag{1.17}$$

respectively. Differentiating these two equations with respect to h, and set the equations equal to zero, we find an optimal step size of $h \sim 10^{-5}$ for equation (1.16), which gives an error of $R_2 \sim 10^{-16}/10^{-5} + (10^{-5})^2/6 \simeq 10^{-10}$, and $h \sim 10^{-4}$ for equation (1.17), which gives an error of $R_4 \sim 4 \cdot 10^{-16}/(10^{-4})^2 + (10^{-4})^2/12 \simeq 10^{-8}$. Note that we get the surprising result for the first order derivative in equation (1.14), that a higher step size gives a more accurate result.

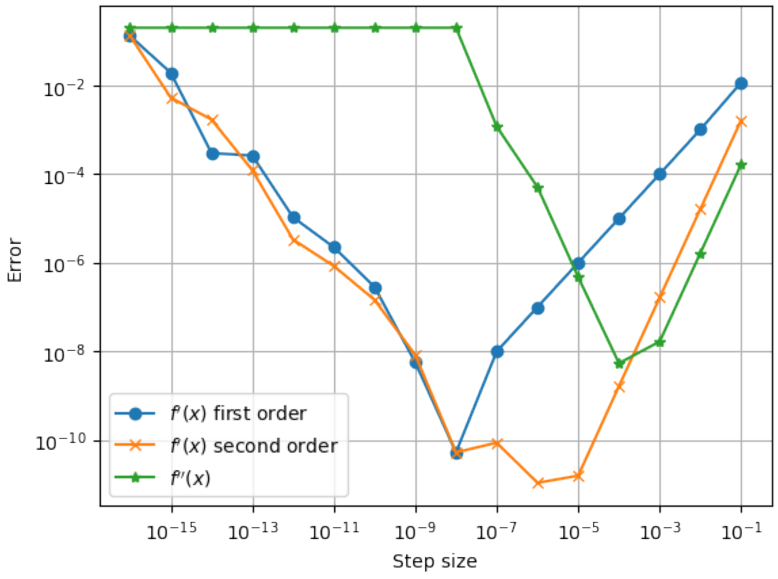

Fig. 1.4 Error in the numerical derivative and second derivative of $\sin x$ at $x = 0.2$ for different step size.

2 Linear and nonlinear equations

Solving systems of equations are one of the most common tasks that we use computers for within modeling. A typical task is that we have have a model that contains a set of unknown parameters which we want to determine. To determine these parameters we need to solve a set of equations. In many cases these equations are nonlinear, but often a nonlinear problem is solved *by linearize* the nonlinear equations, and thereby reducing it to a sequence of linear algebra problems. Thus the topic of solving linear systems of equations have been extensively studied, and sophisticated linear equation solving packages have been developed. Python uses functions from the LAPACK[1] library. In this course we will only cover the theory behind numerical linear algebra superficially, and the main purpose is to shed some light on some of the challenges one might encounter solving linear systems. In particular it is important for you to understand when it is stated in the NumPy documentation that the standard linear solver: `solve`[2] function uses *LU-decomposition* and *partial pivoting*.

After covering some basics of numerical linear algebra, we will shift focus to nonlinear equations. Contrary to linear equations, you will most likely find that the functions available in various Python library will *not* cover your needs and in many cases fail to give you the correct solution. The reason for this is that the solution of a nonlinear equation is greatly

[1] `https://en.wikipedia.org/wiki/LAPACK`
[2] `https://docs.scipy.org/doc/numpy/reference/generated/numpy.linalg.solve.html`

dependent on the starting point, and a combination of various techniques must be used.

2.1 Solving linear equations

There are a number of excellent books covering this topic, see e.g. [1, 4, 2, 3]. In most of the examples covered in this course we will encounter problems where we have a set of *linearly independent* equations and one equation for each unknown. For these type of problems there are a number of methods that can be used, and they will find a solution in a finite number of steps. If a solution cannot be found it is usually because the equations are not linearly independent, and our formulation of the physical problem is wrong.

Assume that we would like to solve the following set of equations:

$$2x_0 + x_1 + x_2 + 3x_3 = 1, \tag{2.1}$$

$$x_0 + x_1 + 3x_2 + x_3 = -3, \tag{2.2}$$

$$x_0 + 4x_1 + x_2 + x_3 = 2, \tag{2.3}$$

$$x_0 + x_1 + x_2 + x_3 = 1. \tag{2.4}$$

These equations can be written in matrix form as:

$$\mathbf{A} \cdot \mathbf{x} = \mathbf{b}, \tag{2.5}$$

where:

$$\mathbf{A} \equiv \begin{pmatrix} 2 & 1 & 1 & 3 \\ 1 & 1 & 3 & 1 \\ 1 & 4 & 1 & 1 \\ 1 & 1 & 2 & 2 \end{pmatrix} \qquad \mathbf{b} \equiv \begin{pmatrix} 1 \\ -3 \\ 2 \\ 1 \end{pmatrix} \qquad \mathbf{x} \equiv \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix}. \tag{2.6}$$

You can easily verify that $x_0 = -4, x_1 = 1, x_2 = -1, x_3 = 3$ is the solution to the above equations by direct substitution. If we were to replace one of the above equations with a linear combination of any of the other equations, e.g. replace equation (2.4) with $3x_0 + 2x_1 + 4x_2 + 4x_3 = -2$, there would be no solution. This can be checked by calculating the determinant of the matrix $\mathbf{A}$, if $\det \mathbf{A} = 0$, What is the difficulty in solving these equations? Clearly if none of the equations are linearly dependent, and we have N independent linear equations, it should be straight forward to solve them? Two major numerical problems are i) even if the equations are not exact linear combinations of each other, they could be very close, and as the numerical algorithm progresses they could at some stage become linearly dependent due to roundoff errors. ii) roundoff errors may accumulate if the number of equations are large [1].

2.1.1 Gauss-Jordan elimination

Let us continue the discussion by consider Gauss-Jordan elimination, which is a *direct* method. A direct method uses a final set of operations to obtain a solution. According to [1] Gauss-Jordan elimination is the method of choice if we want to find the inverse of $\mathbf{A}$. However, it is slow when it comes to calculate the solution of equation (2.5). Even if speed and memory use is not an issue, it is also not advised to first find the inverse, $\mathbf{A}^{-1}$, of $\mathbf{A}$, then multiply it with $\mathbf{b}$ to obtain the solution, due to roundoff errors (Roundoff errors occur whenever we subtract to numbers that are very close to each other). To simplify our notation, we write equation (2.6) as:

$$\left(\begin{array}{cccc|c} 2&1&1&3&1\\ 1&1&3&1&-3\\ 1&4&1&1&2\\ 1&1&2&2&1 \end{array}\right). \tag{2.7}$$

The numbers to the left of the vertical dash is the matrix $\mathbf{A}$, and to the right is the vector $\mathbf{b}$. The Gauss-Jordan elimination procedure proceeds by doing the same operation on the right and left side of the dash, and the goal is to get only zeros on the lower triangular part of the matrix. This is achieved by multiplying rows with the same (nonzero) number, swapping rows, adding a multiple of a row to another:

$$\left(\begin{array}{cccc|c} 2&1&1&3&1\\ 1&1&3&1&-3\\ 1&4&1&1&2\\ 1&1&2&2&1 \end{array}\right) \to \left(\begin{array}{cccc|c} 2&1&1&3&1\\ 0&1/2&5/2&-1/2&-7/2\\ 0&7/2&1/2&-1/2&3/2\\ 0&1/2&3/2&1/2&1/2 \end{array}\right) \to \tag{2.8}$$
$$\left(\begin{array}{cccc|c} 2&1&1&3&1\\ 0&1/2&5/2&-1/2&-7/2\\ 0&0&-17&3&26\\ 0&0&1&-1&4 \end{array}\right) \to \left(\begin{array}{cccc|c} 2&1&1&3&1\\ 0&1/2&5/2&-1/2&-7/2\\ 0&0&-17&3&26\\ 0&0&0&14/17&42/17 \end{array}\right)$$

The operations done are: $(1 \to 2)$ multiply first row with $-1/2$ and add to second, third and the fourth row, $(2 \to 3)$ multiply second row with -7, and add to third row, multiply second row with -1 and add to fourth row, $(3 \to 4)$ multiply third row with $-1/17$ and add to fourth row. These operations can easily be coded into Python:

```
A = np.array([[2, 1, 1, 3,],[1, 1, 3, 1],
              [1, 4, 1, 1, ],[1, 1, 2, 2 ]],float)
b = np.array([1,-3,2,1],float)
N=4
# Gauss-Jordan Elimination
for i in range(1,N):
    fact    = A[i:,i-1]/A[i-1,i-1]
    A[i:,] -= np.outer(fact,A[i-1,])
    b[i:]  -= b[i-1]*fact
```

Notice that the final matrix has only zeros beyond the diagonal, such a matrix is called *upper triangular*. We still have not found the final solution, but from an upper triangular (or lower triangular) matrix it is trivial to determine the solution. The last row immediately gives us $14/17z = 42/17$ or $z = 3$, now we have the solution for z and the next row gives: $-17y + 3z = 26$ or $y = (26 - 3 \cdot 3)/(-17) = -1$, and so on. In a more general form, we can write our solution of the matrix $\mathbf{A}$ after making it upper triangular as:

$$\begin{pmatrix} a'_{0,0} & a'_{0,1} & a'_{0,2} & a'_{0,3} \\ 0 & a'_{1,1} & a'_{1,2} & a'_{1,3} \\ 0 & 0 & a'_{2,2} & a'_{2,3} \\ 0 & 0 & 0 & a'_{3,3} \end{pmatrix} \cdot \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} b'_0 \\ b'_1 \\ b'_2 \\ b'_3 \end{pmatrix} \tag{2.9}$$

The backsubstitution can then be written formally as:

$$x_i = \frac{1}{a'_{ii}} \left[b'_i - \sum_{j=i+1}^{N-1} a'_{ij} x_j \right], \quad i = N-1, N-2, \ldots, 0 \tag{2.10}$$

The backsubstitution can now easily be implemented in Python as:

```
# Backsubstitution
sol = np.zeros(N,float)
sol[N-1]=b[N-1]/A[N-1,N-1]
for i in range(2,N+1):
    sol[N-i]=(b[N-i]-np.dot(A[(N-i),],sol))/A[N-i,N-i]
```

Notice that in the Python implementation, we have used vector operations instead of for loops. This makes the code more efficient, but it could also be implemented with for loops:

```
# Backsubstitution - for loop
sol = np.zeros(N,float)
for i in range(N-1,-1,-1):
    sol[i]= b[i]
    for j in range(i+1,N):
        sol[i] -= A[i][j]*sol[j]
    sol[i] /= A[i][i]
```

There are at least two things to notice with our implementation:

- Matrix and vector notation makes the code more compact and efficient. In order to understand the implementation it is advised to put $i = 1, 2, 3, 4$, and then execute the statements in the Gauss-Jordan elimination and compare with equation (2.8).
- The implementation of the Gauss-Jordan elimination is not robust, in particular one could easily imagine cases where one of the leading coefficients turned out as zero, and the routine would fail when we divide by `A[i-1,i-1]`. By simply changing equation (2.2) to $2x_0 + x_1 + 3x_2 + x_3 = -3$, when doing the first Gauss-Jordan elimination, both x_0 and x_1 would be canceled. In the next iteration we try to divide next equation by the leading coefficient of x_1, which is zero, and the whole procedure fails.

2.1.2 Pivoting

The solution to the last problem is solved by what is called *pivoting*. The element that we divide on is called the *pivot element*. It actually turns out that even if we do Gauss-Jordan elimination *without* encountering a zero pivot element, the Gauss-Jordan procedure is numerically unstable in the presence of roundoff errors [1]. There are two versions of pivoting, *full pivoting* and *partial pivoting*. In partial pivoting we only interchange rows, while in full pivoting we also interchange rows and columns. Partial pivoting is much easier to implement, and the algorithm is as follows:

1. Find the row in $\mathbf{A}$ with largest absolute value in front of x_0 and change with the first equation, switch corresponding elements in $\mathbf{b}$
2. Do one Gauss-Jordan elimination, find the row in $\mathbf{A}$ with the largest absolute value in front of x_1 and switch with the second (same for $\mathbf{b}$), and so on.

For a linear equation we can multiply with a number on each side and the equation would be unchanged, so if we where to multiply one of the equations with a large value, we are almost sure that this equation would be placed first by our algorithm. This seems a bit strange as our mathematical problem is the same. Sometimes the linear algebra routines tries to normalize the equations to find the pivot element that would have been the largest element if all equations were normalized according to some rule, this is called *implicit pivoting*.

2.1.3 LU decomposition

As we have already seen, if the matrix **A** is reduced to a triangular form it is trivial to calculate the solution by using backsubstitution. Thus if it was possible to decompose the matrix **A** as follows:

$$\mathbf{A} = \mathbf{L} \cdot \mathbf{U} \tag{2.11}$$

$$\begin{pmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,0} & a_{2,0} & a_{2,2} & a_{2,3} \\ a_{3,0} & a_{3,0} & a_{3,0} & a_{3,3} \end{pmatrix} = \begin{pmatrix} l_{0,0} & 0 & 0 & 0 \\ l_{1,0} & l_{1,1} & 0 & 0 \\ l_{2,0} & l_{2,0} & l_{2,2} & 0 \\ l_{3,0} & l_{3,0} & l_{3,0} & l_{3,3} \end{pmatrix} \cdot \begin{pmatrix} u_{0,0} & u_{0,1} & u_{0,2} & u_{0,3} \\ 0 & u_{1,1} & u_{1,2} & u_{1,3} \\ 0 & 0 & u_{2,2} & u_{2,3} \\ 0 & 0 & 0 & u_{3,3} \end{pmatrix}.$$

The solution procedure would then be to rewrite equation (2.5) as:

$$\mathbf{A} \cdot \mathbf{x} = \mathbf{L} \cdot \mathbf{U} \cdot \mathbf{x} = \mathbf{b}, \tag{2.12}$$

If we define a new vector **y**:

$$\mathbf{y} \equiv \mathbf{U} \cdot \mathbf{x}, \tag{2.13}$$

we can first solve for the **y** vector:

$$\mathbf{L} \cdot \mathbf{y} = \mathbf{b}, \tag{2.14}$$

and then for **x**:

$$\mathbf{U} \cdot \mathbf{x} = \mathbf{y}. \tag{2.15}$$

Note that the solution to equation (2.14) would be done by *forward substitution*:

$$y_i = \frac{1}{l_{ii}} \left[b_i - \sum_{j=0}^{i-1} l_{ij} x_j \right], \quad i = 1, 2, \ldots N-1. \tag{2.16}$$

Why go to all this trouble? First of all it requires (slightly) less operations to calculate the LU decomposition and doing the forward and backward substitution than the Gauss-Jordan procedure discussed earlier. Secondly, and more importantly, is the fact that in many cases one would like to calculate the solution for different values of the **b** vector in equation

(2.12). If we do the LU decomposition first we can calculate the solution quite fast using backward and forward substitution for any value of the **b** vector.

The NumPy function `solve`[3], uses LU decomposition and partial pivoting, and we can find the solution to our previous problem simply by the following code:

```
from numpy.linalg import solve
x=solve(A,b)
```

2.2 Example: Linear regression

In the previous section, we considered a system of N equations and N unknown $(x_0, x_1, \ldots, x_N)$. In general we might have more equations than unknowns or more unknowns than equations. An example of the former is linear regression, we might have many data points and we would like to fit a line through the points. How do you fit a single lines to more than two points that does not line on the same line? One way to do it is to minimize the distance from the line to the points, as illustrated in figure 2.1.

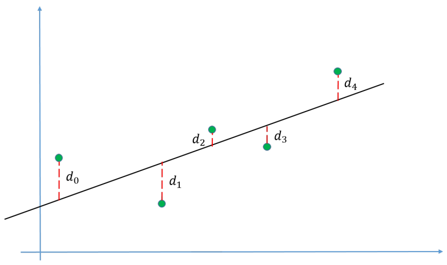

Fig. 2.1 Linear regression by minimizing the total distance to all the points.

Mathematically we can express the distance between a data point (x_i, y_i) and the line $f(x)$ as $y_i - f(x_i)$. Note that this difference can be negative or positive depending if the data point lies below or above the line. We can then take the absolute value of all the distances, and try to minimize them. When we minimize something we take the derivative of the expression and put it equal to zero. As you might remember from Calculus it is extremely hard to work with the derivative of the absolute

[3] https://docs.scipy.org/doc/numpy/reference/generated/numpy.linalg.solve.html

value, because it is discontinuous. A much better approach is to square each distance and sum them:

$$S = \sum_{i=0}^{3} (y_i - f(x_i))^2 = \sum_{i=0}^{3} (y_i - a_0 - a_1 x_i)^2. \tag{2.17}$$

This is the idea behind *least square*, and linear regression. One thing you should be aware of is that points lying far from the line will contribute more to equation (2.17). The underlying assumption is that each data point provides equally precise information about the process, this is often not the case. When analyzing experimental data, there may be points deviating from the expected behaviour, it is then important to investigate if these points are more affected by measurements errors than the others. If that is the case one should give them less weight in the least square estimate, by extending the formula above:

$$S = \sum_{i=0}^{3} \omega_i (y_i - f(x_i))^2 = \sum_{i=0}^{3} \omega_i (y_i - a_0 - a_1 x_i)^2, \tag{2.18}$$

ω_i is a weight factor.

Let us continue with equation (2.17)

Two great explanations linear regression by matrices[4], and R^2-squared[5]

2.3 Example: Solving the heat equation regression

2.4 Singular Value Decomposition

2.5 QR factorization

2.6 Solving nonlinear equations

The purpose of this section is to introduce a handful of techniques for solving a nonlinear equation. In many cases a combination of methods must be used, and the algorithm must be adopted to your specific problem.

[4] https://medium.com/@andrew.chamberlain/the-linear-algebra-view-of-least-squares-regression

[5] https://medium.com/@andrew.chamberlain/a-more-elegant-view-of-r-squared-a0a14c177dc3

3 Numerical integration

3.1 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.

3.1.1 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 3.1. The area of one box is:

$$M(x_k, x_k + h) = f(x_k + \frac{h}{2})h, \tag{3.1}$$

and the area of all the boxes is:

$$\begin{aligned} I(a,b) &= \int_a^b f(x)dx \simeq \sum_{k=0}^{N-1} M(x_k, x_k + h) \\ &= h \sum_{k=0}^{N-1} f(x_k + \frac{h}{2}) = h \sum_{k=0}^{N-1} f(a + (k + \frac{1}{2})h). \end{aligned} \tag{3.2}$$

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 $k = 0$.

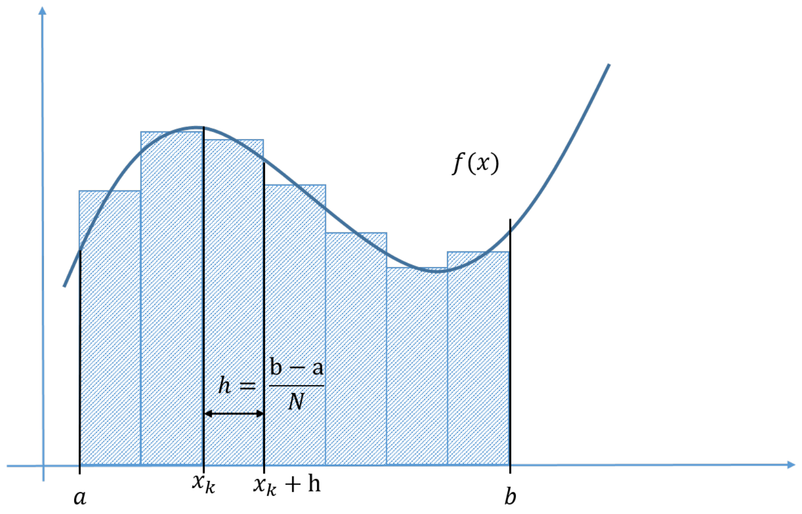

Fig. 3.1 Integrating a function with the midpoint rule.

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(func, lower_limit, upper_limit,N):
    """ calculates the area of func over the domain lower_limit
        to upper limit using N integration points """
    h    = (upper_limit-lower_limit)/N # step size
    area = 0.
    for all integration points do:
        estimate integration midpoint value, xi
        add area under curve: func(xi)*h
    return area
```

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
```

3.1.2 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 3.2.

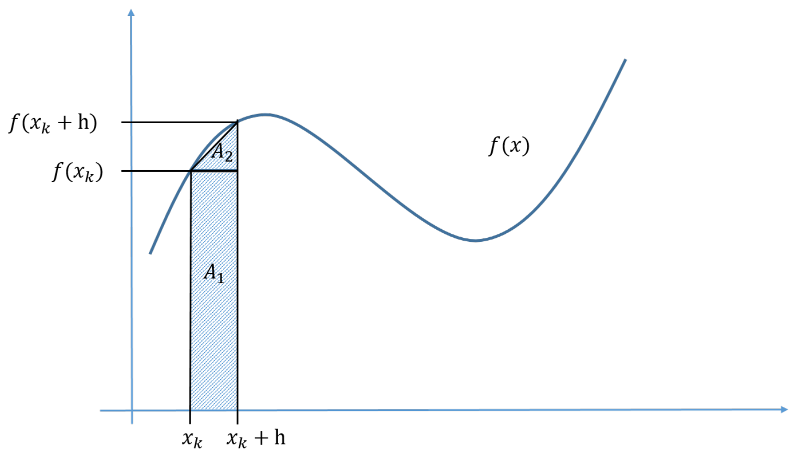

Fig. 3.2 Integrating a function with the trapezoidal rule.

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:

$$A \equiv T(x_k, x_k+h) = (f(x_k+h) + f(x_k))h/2. \tag{3.3}$$

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

$$\begin{aligned} I(a,b) = \int_a^b f(x)dx &\simeq \frac{1}{2}h\sum_{k=0}^{N-1}[f(x_k+kh)+f(x_k)] \\ &= h\left[\frac{1}{2}f(a)+f(a+h)+f(a+2h)+\right. \\ &\quad \left.\cdots+f(a+(N-2)h)+\frac{1}{2}f(b)\right] \\ &= h\left[\frac{1}{2}f(a)+\frac{1}{2}f(b)+\sum_{k=1}^{N-1}f(a+kh)\right]. \end{aligned} \tag{3.4}$$

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:

```python
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(func, lower_limit, upper_limit,N):
    """ calculates the area of func over the domain lower_limit
        to upper limit using N integration points """
    h       = (upper_limit-lower_limit)/N # step size
    area    = 0.5*(func(lower_limit)+func(upper_limit))
    val     = lower_limit
    for all integration points do:
        estimate integration midpoint value, xi
        add are under curve: func(xi)*h
    return area
```

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 3.1, 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.

3.1.3 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 $x_k + h$:

$$\begin{aligned} f(x) &= f(x_k + h/2) + f'(x_k + h/2)(x - (x_k + h/2)) \\ &+ \frac{1}{2!}f''(x_k + h/2)(x - (x_k + h/2))^2 + \mathcal{O}(h^3) \end{aligned} \tag{3.5}$$

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

$$I(x_k, x_k + h) = \int_{x_k}^{x_k+h} \Big[f(x_k + h/2) + f'(x_k + h/2)(x - (x_k + h/2)) + \frac{1}{2!}f''(x_k + h/2)(x - (x_k + h/2))^2 + \mathcal{O}(h^3)\Big]\, dx \tag{3.6}$$

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

$$\begin{aligned} I(x_k, x_k + h) &= f(x_k + h/2) \cdot h + f'(x_k + h/2) \int_0^h (u - h/2)du \\ &+ \frac{1}{2}f''(x_k + h/2) \int_0^h (u - h/2)^2 du + \mathcal{O}(h^4) \\ &= f(x_k + h/2) \cdot h - \frac{h^3}{24}f''(x_k + h/2) + \mathcal{O}(h^4). \end{aligned} \tag{3.7}$$

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:

$$E_{M,k} = I(x_k, x_k + h) - f(x_k + h/2) \cdot h = -\frac{h^3}{24}f''(x_k + h/2), \tag{3.8}$$

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 $[a, b]$, $f''(\eta)$:

$$E_M = \sum_{k=0}^{N-1} E_{M,k} = -\frac{h^3}{24}\sum_{k=0}^{N-1} f''(x_k + h/2) \leq -\frac{Nh^3}{24}f''(\eta), \tag{3.9}$$

$$E_M \leq -\frac{(b-a)^3}{24N^2}f''(\eta), \tag{3.10}$$

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:

$$E_T = \frac{(b-a)^3}{12N^2}f''(\overline{\eta}). \tag{3.11}$$

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 (3.9), we see that it is closely related to the midpoint rule for $\int_a^b f''(x)dx$, hence:

$$E_M = -\frac{h^2}{24}h\sum_{k=0}^{N-1} f''(x_k + h/2) \simeq -\frac{h^2}{24}\int_a^b f''(x)dx \tag{3.12}$$

$$E_M \simeq \frac{h^2}{24}\left[f'(b) - f'(a)\right] = -\frac{(b-a)^2}{24N^2}\left[f'(b) - f'(a)\right] \tag{3.13}$$

The corresponding formula for the trapezoid formula is:

$$E_T \simeq \frac{h^2}{12}\left[f'(b) - f'(a)\right] = \frac{(b-a)^2}{12N^2}\left[f'(b) - f'(a)\right] \tag{3.14}$$

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(func, lower_limit, upper_limit, tol):
    """ calculates the area of func over the domain lower_limit
        to upper limit for the specified tolerance tol """
    dfa  = df(lower_limit,func) # derivative in point a
    dfb  = df(upper_limit,func) # derivative in point b
    h    = np.sqrt(abs(24*tol/(dfb-dfa)))
    print('Number of intervals = ', (upper_limit-lower_limit)/h)
    for all integration points do:
        estimate integration midpoint value, xi
        add area under curve: func(xi)*h
    return area
```

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}.

3.1.4 Practical Estimation of Errors on Integrals

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:

$$I_2 - I_1 = I - ch_2^2 - (I - ch_1^2) = 3ch_2^2, \tag{3.15}$$

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

$$E(a,b) = ch_2^2 = \frac{1}{3}(I_2 - I_1). \tag{3.16}$$

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:

$$I_2(a,b) = h_2\left[\frac{1}{2}f(a)+\frac{1}{2}f(b)+\sum_{k=1}^{N_2-1}f(a+kh_2)\right], \tag{3.17}$$

$$= h_2\left[\frac{1}{2}f(a)+\frac{1}{2}f(b)+\sum_{k=\text{even values}}^{N_2-1}f(a+kh_2)\right.$$
$$\left.+\sum_{k=\text{odd values}}^{N_2-1}f(a+kh_2)\right], \tag{3.18}$$

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

$$\sum_{k=\text{even values}}^{N_2-1}f(a+kh_2)=\sum_{k=0}^{N_1-1}f(a+2kh_2)=\sum_{k=0}^{N_1-1}f(a+kh_1), \tag{3.19}$$

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

$$I_2(a,b) = h_2\left[\frac{1}{2}f(a)+\frac{1}{2}f(b)+\sum_{k=0}^{N_1-1}f(a+kh_1)\right.$$
$$\left.+\sum_{k=\text{odd values}}^{N_2-1}f(a+kh_2)\right] \tag{3.20}$$

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

$$I_2(a,b)=\frac{1}{2}I_1(a,b)+h_2\sum_{k=\text{odd values}}^{N_2-1}f(a+kh_2) \tag{3.21}$$

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:

```python
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(func, lower_limit, upper_limit, tol):
    """ calculates the area of func over the domain lower_limit
        to upper limit for the specified tolerance tol """
    h       = (upper_limit-lower_limit)
    area    = 0.5*(func(lower_limit)+func(upper_limit))
    calc_tol = tol + 1 # just larger than tol to enter the while loop
    while(calc_tol>tol):
        half the step size h /= 2
        for all odd integration points in the domain:
            sum up all the odd function values in odd_terms
        new_area = 0.5*area + h*odd_terms
        calc_tol = abs(new_area-area)/3
        area     = new_area # store new values for next iteration
    print('Number of intervals = ', (upper_limit-lower_limit)/h)
    return area #while loop ended and we can return the area
```

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.

3.2 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] : $I = I_i + ch_i^2 + \mathcal{O}(h^4)$. 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:

$$I = I_{i+1} + ch^2 + \mathcal{O}(h^4) = I_{i+1} + \frac{1}{3}\left[I_{i+1} - I_i\right] + \mathcal{O}(h^4). \tag{3.22}$$

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

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

use these two estimates to calculate the error term:

$$I_{i+1} - I_i = I - ch_{i+1}^4 - (I - ch_i^4) = -ch_{i+1}^4 + c(2h_{i+1})^4 = 15ch_{i+1}^4,$$
$$ch_{i+1}^4 = \frac{1}{15}\left[I_{i+1} - I_i\right] + \mathcal{O}(h^6). \tag{3.23}$$

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:

$$I = I_{i+1} + \frac{1}{15}\left[I_{i+1} - I_i\right] + \mathcal{O}(h^6). \tag{3.24}$$

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 (3.24), can now be written:

$$I = R_{i+1,2} + \frac{1}{15}\left[R_{i+1,2} - R_{i,2}\right] + \mathcal{O}(h^6). \tag{3.25}$$

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

$$I = R_{i+1,m+1} + c_m h_i^{2m+2} + \mathcal{O}(h_i^{2m+4}) \tag{3.26}$$
$$\begin{aligned} I &= R_{i,m+1} + c_m h_{i-1}^{2m+2} + \mathcal{O}(h_{i-1}^{2m+4}) \\ &= R_{i,m+1} + 2^{2m+2} c_m h_i^{2m+2} + \mathcal{O}(h_{i-1}^{2m+4}), \end{aligned} \tag{3.27}$$

where, as before $h_{i-1} = 2h_i$. Subtracting equation (3.26) and (3.27), we find an expression for the error term:

$$c_m h_i^{2m+2} = \frac{1}{4^{m+1} - 1}(R_{i,m} - R_{i-1,m}) \tag{3.28}$$

Then the estimate for the integral in equation (3.27) is:

$$I = R_{i,m+1} + \mathcal{O}(h_i^{2m+2}) \tag{3.29}$$
$$R_{i,m+1} = R_{i,m} + \frac{1}{4^{m+1} - 1}(R_{i+1,m} - R_{i,m}). \tag{3.30}$$

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 3.3.

$$\begin{tikzcd}
I_0 = R_{0,0} \arrow[dr] & & & & \\
I_1 = R_{1,0} \arrow[r] \arrow[dr] & R_{1,1} \arrow[dr] & & & \\
I_2 = R_{2,0} \arrow[r] \arrow[dr] & R_{2,1} \arrow[r] \arrow[dr] & R_{2,2} \arrow[dr] & & \\
I_3 = R_{3,0} \arrow[r] \arrow[dr] & R_{3,1} \arrow[r] \arrow[dr] & R_{3,2} \arrow[r] \arrow[dr] & R_{3,3} \arrow[dr] & \\
I_4 = R_{4,0} \arrow[r] & R_{4,1} \arrow[r] & R_{4,2} \arrow[r] & R_{4,3} \arrow[r] & R_{4,4}
\end{tikzcd}$$

Fig. 3.3 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:

```python
import numpy as np
# Function to be integrated
def f(x):
    return np.sin(x)
# step size is choosen automatically to reach (at least)
# the specified tolerance
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 all odd terms 1, 3, ..., N-1 do:
            evaulate function values at odd points,
            sum them and store in odd_terms
        R[i,0]   = 0.5*R[i-1,0] + h*odd_terms

        for all m in 0, 1, ..., i-1:
            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 estimated tolerance calc_tol is lower than tol:
            break  # estimated precision reached
        if max number of iterations are reached (i == Nmax-1):
            print('Romberg routine did not converge after ',
              Nmax, 'iterations!')

    if(show==True):
        print out all triangualar elements in R[i,m]

    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.

3.2.1 Gaussian Quadrature

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

$$\int_a^b f(x)dx = \sum_{k=0}^{N-1} \omega_k f(x_k), \tag{3.31}$$

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 $[-1,1]$:

$$\int_a^b f(t)dt = \frac{b-a}{2}\int_{-1}^{1} f(x)dx \text{ , where:} \tag{3.32}$$

$$x = \frac{2}{b-a}t - \frac{b+a}{b-a}. \tag{3.33}$$

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 (3.31):

$$\int_{-1}^{1} f(x)\,dx \simeq \omega_0 f(x_0). \tag{3.34}$$

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:

$$\int_{-1}^{1} f(x)\,dx = \int_{-1}^{1} 1\,dx = 2 = \omega_0, \tag{3.35}$$

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

$$\int_{-1}^{1} f(x)\,dx = \int_{-1}^{1} x\,dx = 0 = \omega_0 f(x_0) = 2x_0, \tag{3.36}$$

hence $x_0 = 0$.

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

$$\int_{-1}^{1} f(x)\,dx \simeq 2f(0), \text{ or:}$$

$$\int_{a}^{b} f(t)\,dt \simeq \frac{b-a}{2}2f(\frac{b+a}{2}) = (b-a)f(\frac{b+a}{2}). \tag{3.37}$$

This equation is equal to the midpoint rule, by choosing $b = a + h$ we reproduce equation (3.1). If we choose $N = 2$:

$$\int_{-1}^{1} f(x)\,dx \simeq \omega_0 f(x_0) + \omega_1 f(x_1), \tag{3.38}$$

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

$$\int_{-1}^{1} 1\,dx = 2 = \omega_0 f(x_0) + \omega_1 f(x_1) = \omega_0 + \omega_1\,, \tag{3.39}$$

$$\int_{-1}^{1} x\,dx = 0 = \omega_0 f(x_0) + \omega_1 f(x_1) = \omega_0 x_0 + \omega_1 x_1\,, \tag{3.40}$$

$$\int_{-1}^{1} x^2\,dx = \frac{2}{3} = \omega_0 f(x_0) + \omega_1 f(x_1) = \omega_0 x_0^2 + \omega_1 x_1^2\,, \tag{3.41}$$

$$\int_{-1}^{1} x^3\,dx = 0 = \omega_0 f(x_0) + \omega_1 f(x_1) = \omega_0 x_0^3 + \omega_1 x_1^3\,, \tag{3.42}$$

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:

$$\int_{-1}^{1} f(x)\,dx \simeq f(-\frac{1}{\sqrt{3}}) + f(\frac{1}{\sqrt{3}})\,, \text{ or:} \tag{3.43}$$

$$\int_{a}^{b} f(x)\,dx \simeq \frac{b-a}{2}\left[f(-\frac{b-a}{2}\frac{1}{\sqrt{3}} + \frac{b+a}{2}) + f(\frac{b-a}{2}\frac{1}{\sqrt{3}} + \frac{b+a}{2})\right]. \tag{3.44}$$

```python
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:

$$\int_{-1}^{1} 1\,dx = 2 = \omega_0 + \omega_1 + \omega_2\,, \tag{3.45}$$

$$\int_{-1}^{1} x\,dx = 0 = \omega_0 x_0 + \omega_1 x_1 + \omega_2 x_2\,, \tag{3.46}$$

$$\int_{-1}^{1} x^2\,dx = \frac{2}{3} = \omega_0 x_0^2 + \omega_1 x_1^2 + \omega_2 x_2^2\,, \tag{3.47}$$

$$\int_{-1}^{1} x^3\,dx = 0 = \omega_0 x_0^3 + \omega_1 x_1^3 + \omega_2 x_2^3\,, \tag{3.48}$$

$$\int_{-1}^{1} x^4\,dx = \frac{2}{5} = \omega_0 x_0^4 + \omega_1 x_1^4 + \omega_2 x_2^4\,, \tag{3.49}$$

$$\int_{-1}^{1} x^5\,dx = 0 = \omega_0 x_0^5 + \omega_1 x_1^5 + \omega_2 x_2^5\,, \tag{3.50}$$

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:

```python
def int_gaussquad2(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 $11^2 = 121$.

3.2.2 Error term on Gaussian Integration

The Gaussian integration rule of order N integrates exactly a polynomial of order $2N-1$. From Taylors error formula, see equation (1.4) in Chapter 1, we can easily see that the error term must be of order $2N$, and be proportional to $f^{(2N)}(\eta)$, see [2] 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 3.1.4. 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:

$$I = I_N + ch_N^{2N}, \tag{3.51}$$

$$I = I_{N+1} + ch_{N+1}^{2N+1}. \tag{3.52}$$

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 (3.51) and (3.52) we can show that a reasonable estimate for the error term ch^{2N} would be:

$$ch^N = I_{N+1} - I_N. \tag{3.53}$$

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]

3.2.3 Common Weight functions for Classical Gaussian Quadratures

3.2.4 Which method to use in a specific case?

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:

[2] `https://docs.scipy.org/doc/scipy-0.14.0/reference/generated/scipy.integrate.quadrature.html`

$$I(a,b) = \int_a^b f(x)dx \simeq \sum_{k=0}^{N-1} M(x_k, x_k+h) = \sum_{k=0}^{N-1} h_i f(x_k + \frac{h_i}{2}) \quad (3.54)$$

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 3.1: 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 $E_T = -2E_M$.

$$\begin{aligned} I(x_k, x_k+h) &= M(x_k, x_k+h) + E_M \text{ (midpoint rule)} && (3.55)\\ I(x_k, x_k+h) &= T(x_k, x_k+h) + E_T \\ &= T(x_k, x_k+h) - 2E_M \text{ (trapezoidal rule)}, && (3.56) \end{aligned}$$

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

$$3I(x_k,x_k+h) = 2M(x_k,x_k+h)+T(x_k,x_k+h) \tag{3.57}$$
$$= 2f(x_k+\frac{h}{2})h + [f(x_k+h)+f(x_k)]\,\frac{h}{2} \tag{3.58}$$
$$I(x_k,x_k+h) = \frac{h}{6}\left[f(x_k)+4f(x_k+\frac{h}{2})+f(x_k+h)\right]. \tag{3.59}$$

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

$$I(a,b) = \sum_{k=0}^{N-1} I(x_k,x_k+h)$$
$$= \frac{h}{6}\left[f(a)+4f(a+\frac{h}{2})+2f(a+h)+4f(a+3\frac{h}{2})\right.$$
$$\left. +2f(a+2h)+\cdots+f(b)\right] \tag{3.60}$$
$$= \frac{h'}{3}\left[f(a)+f(b)+4\sum_{k=\text{odd}}^{N-2} f(a+kh')+2\sum_{k=\text{even}}^{N-2} f(a+kh')\right], \tag{3.61}$$

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.

Ordinary differential equations 4

Monte Carlo Methods 5

References

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

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

[3] Gilbert Strang. *Linear Algebra and Learning From Data*. Wellesley-Cambridge Press, 2019.

[4] Lloyd N. Trefethen and David Bau III. *Numerical Linear Algebra*, volume 50. SIAM, 1997.