Modeling and Computational Engineering

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

University of Stavanger

Sep 30, 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

List of Exercises

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

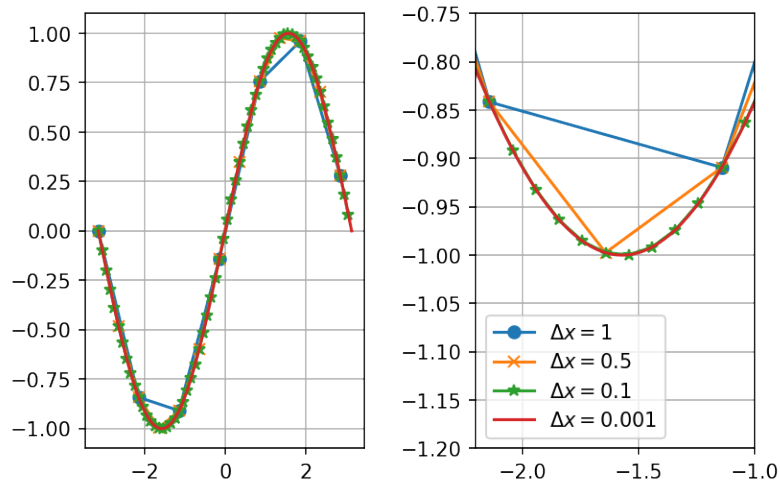

Fig. 1.1 A plot of $\sin x$ for different spacing of the x-values.

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:

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

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`

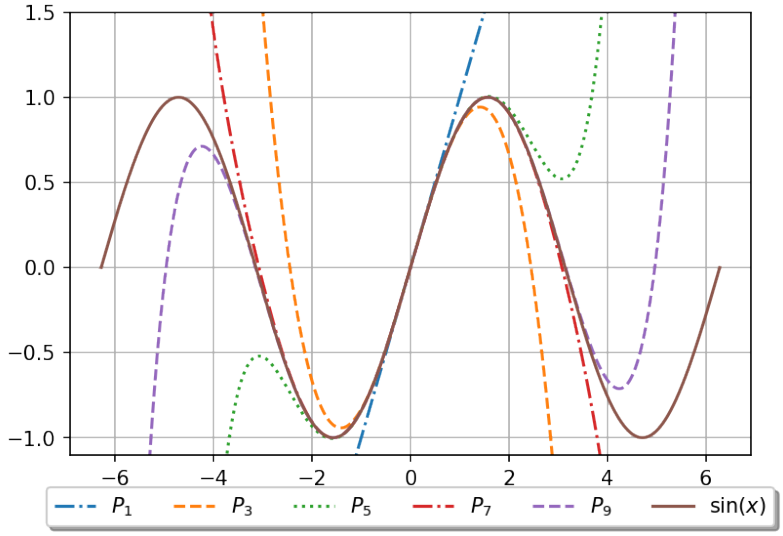

Fig. 1.2 Up to ninth order in the Maclaurin series of $\sin x$.

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:

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

1.2.1 Evaluation of polynomials

How to evaluate a polynomial of the type: $p_n(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n$? 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.

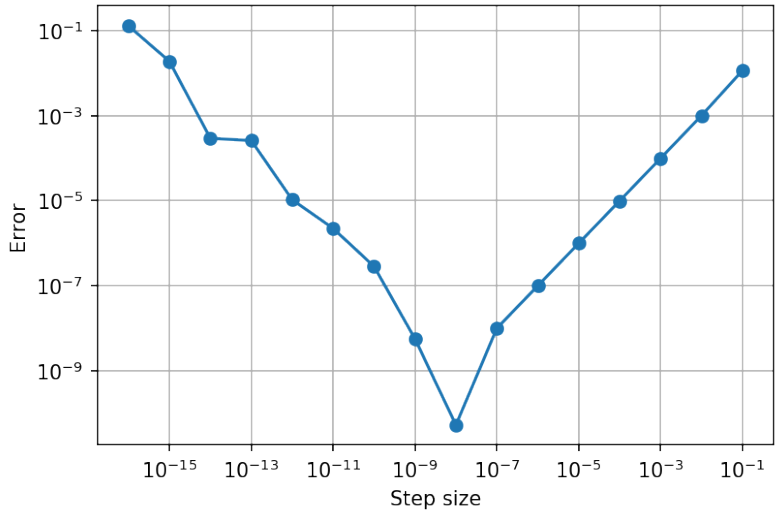

Fig. 1.3 Error in the numerical derivative of $\sin x$ at $x = 0.2$ for different step size.

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:

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

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.

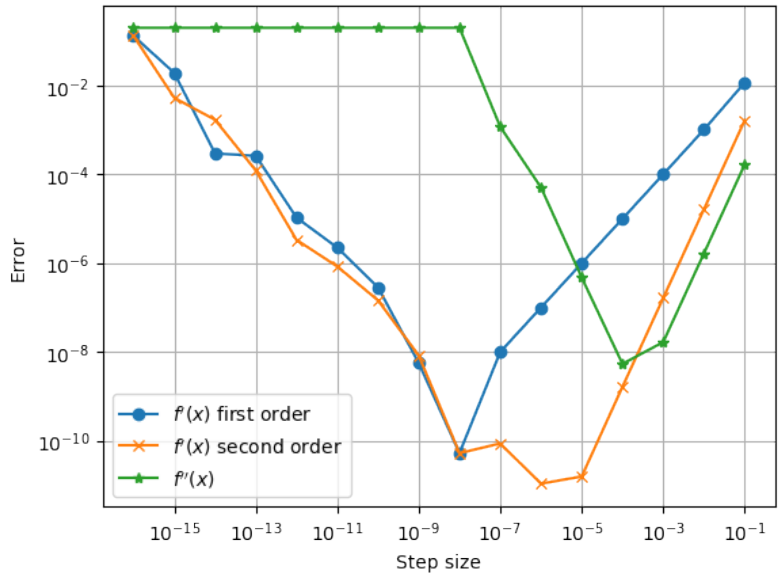

Fig. 1.4 Error in the numerical derivative and second derivative of $\sin x$ at $x = 0.2$ for different step size.

Solving linear systems

2

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 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*.

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

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

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 unique solution (infinite number of solutions). 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:

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

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

The operations done are: $(1 \rightarrow 2)$ multiply first row with $-1/2$ and add to second, third and the fourth row, $(2 \rightarrow 3)$ multiply second row with -7, and add to third row, multiply second row with -1 and add to fourth row, $(3 \rightarrow 4)$ 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 $\mathbf{A}$ 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 $\mathbf{A}$ as follows:

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

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

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 $\mathbf{y}$:

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

we can first solve for the $\mathbf{y}$ vector:

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

and then for $\mathbf{x}$:

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

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 $\mathbf{b}$ 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 $\mathbf{b}$ 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.

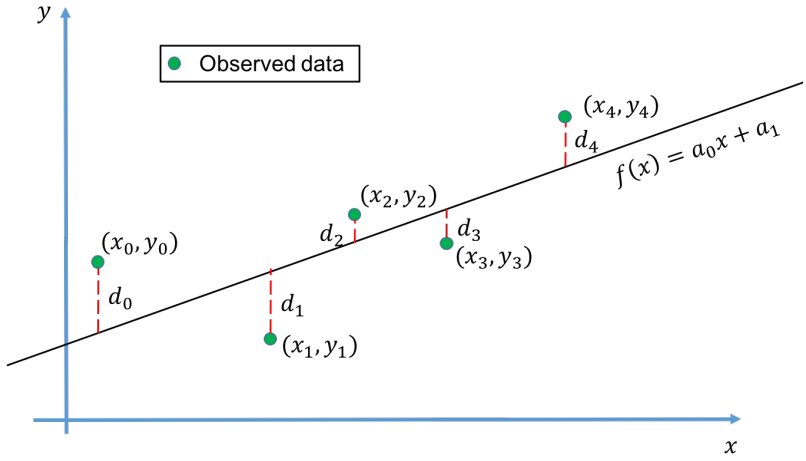

Fig. 2.1 Linear regression by minimizing the total distance to all the points.

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

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

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

$$S = \sum_{i=0}^{N-1} (y_i - f(x_i))^2 = \sum_{i=0}^{N-1} (y_i - a_0 - a_1 x_i)^2. \tag{2.17}$$

(For the example in figure 2.1, $N = 5$.) 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:

$$S = \sum_{i=0}^{N-1} \omega_i (y_i - f(x_i))^2 = \sum_{i=0}^{3} \omega_i (y_i - a_0 - a_1 x_i)^2, \tag{2.18}$$

ω_i is a weight factor.

2.2.1 Solving least square, using algebraic equations

Let us continue with equation (2.17), the algebraic solution is to simply find the value of a_0 and a_1 that minimizes S:

$$\frac{\partial S}{\partial a_0} = -2 \sum_{i=0}^{N-1} (y_i - a_0 - a_1 x_i) = 0, \tag{2.19}$$

$$\frac{\partial S}{\partial a_1} = -2 \sum_{i=0}^{N-1} (y_i - a_0 - a_1 x_i) x_i = 0. \tag{2.20}$$

Defining the mean value as $\overline{x} = \sum_i x_i / N$ and $\overline{y} = \sum_i y_i / N$, we can write equation (2.19) and (2.20) as:

$$\sum_{i=0}^{N-1}(y_i - a_0 - a_1 x_i) = N\overline{y} - a_0 N - a_1 N\overline{x} = 0, \tag{2.21}$$

$$\sum_{i=0}^{N-1}(y_i - a_0 - a_1 x_i)x_i = \sum_i y_i x_i - a_0 N\overline{x} - a_1 \sum_i x_i x_i = 0. \tag{2.22}$$

Solving equation (2.21) with respect to a_0, and inserting the expression into equation (2.22), we find:

$$a_0 = \overline{y} - a_1\overline{x}, \tag{2.23}$$

$$a_1 = \frac{\sum_i y_i x_i - N\overline{xy}}{\sum_i x_i^2 - N\overline{x}^2} = \frac{\sum_i (x_i - \overline{x})(y_i - \overline{y})}{\sum_i (x_i - \overline{x})^2}. \tag{2.24}$$

We leave it as an exercise to show the last expression for a_1. Clearly the equation (2.24) above will in most cases have a solution. But in addition to a solution, it would be good to have an idea of the goodness of the fit. Intuitively it make sense to add all the distances (residuals) d_i in figure 2.1. This is basically what is done when calculating R^2 (R-squared). However, we would also like to compare the R^2 between different datasets. Therefor we need to normalize the sum of residuals, and therefore the following form of the R^2 is used:

$$R^2 = 1 - \frac{\sum_{i=0}^{N-1}(y_i - f(x_i))^2}{\sum_{i=0}^{N-1}(y_i - \overline{y})^2}. \tag{2.25}$$

In python we can implement equation (2.23), (2.24) and (2.25) as:

```
def OLS(x, y):
    # returns regression coefficients
    # in ordinary least square
    # x: observations
    # y: response
    # R^2: R-squared
    n = np.size(x) # number of data points

    # mean of x and y vector
    m_x, m_y = np.mean(x), np.mean(y)

    # calculating cross-deviation and deviation about x
    SS_xy = np.sum(y*x) - n*m_y*m_x
    SS_xx = np.sum(x*x) - n*m_x*m_x

    # calculating regression coefficients
    b_1 = SS_xy / SS_xx
    b_0 = m_y - b_1*m_x
```

```
    #R^2
    y_pred = b_0 + b_1*x
    S_yy   = np.sum(y*y) - n*m_y*m_y
    y_res  = y-y_pred
    S_res  = np.sum(y_res*y_res)

    return(b_0, b_1,1-S_res/S_yy)
```

2.2.2 Least square as a linear algebra problem

It turns out that the least square problem can be formulated as a matrix problem. (Two great explanations see linear regression by matrices[4], and R^2-squared[5].) If we define a matrix $\mathbf{X}$ containing the observations x_i as:

$$\mathbf{X} = \begin{pmatrix} 1 & x_0 \\ 1 & x_1 \\ \vdots & \vdots \\ 1 & x_{N-1} \end{pmatrix}. \tag{2.26}$$

We introduce a vector containing all the response $\mathbf{y}$, and the regression coefficients $\mathbf{a} = (a_0, a_1)$. Then we can write equation (2.18) as a matrix equation:

$$S = (\mathbf{y} - \mathbf{X} \cdot \mathbf{a})^T (\mathbf{y} - \mathbf{X} \cdot \mathbf{a}). \tag{2.27}$$

Note that this equation can easily be extended to more than one observation variable x_i. By simply differentiating equation (2.27) with respect to $\mathbf{a}$, we can show that the derivative has a minimum when:

$$\mathbf{X}^T \mathbf{X} \mathbf{a} = \mathbf{X}^T \mathbf{y} \tag{2.28}$$

Below is a python implementation of equation (2.28).

```
def OLSM(x, y):
    # returns regression coefficients
    # in ordinary least square using solve function
    # x: observations
    # y: response

    XT = np.array([np.ones(len(x)),x],float)
    X  = np.transpose(XT)
    B = np.dot(XT,X)
    C = np.dot(XT,y)
    return solve(B,C)
```

[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

2.3 Sparse matrices and Thomas algorithm

In many practical examples, such as solving partial differential equations the matrices could be quite large and also contain a lot of zeros. A very important class of such matrices are *banded matrices* this is a type of *sparse matrices* containing a lot of zero elements, and the non-zero elements are confined to diagonal bands. In the following we will focus on one important type of sparse matrix the tridiagonal. In the next section we will show how it enters naturally in solving the heat equation. It turns out that solving banded matrices is quite simple, and can be coded quite efficiently. As with the Gauss-Jordan example, lets consider a concrete example:

$$\left(\begin{array}{ccccc|c} b_0 & c_0 & 0 & 0 & 0 & r_0 \\ a_1 & b_1 & c_1 & 0 & 0 & r_1 \\ 0 & a_2 & b_2 & c_2 & 0 & r_2 \\ 0 & 0 & a_3 & b_3 & c_3 & r_3 \\ 0 & 0 & 0 & a_4 & b_4 & r_4 \end{array}\right) \tag{2.29}$$

The right hand side is represented with r_i. The first Gauss-Jordan step is simply to divide by b_0, then we multiply with $-a_1$ and add to second row:

$$\rightarrow \left(\begin{array}{ccccc|c} 1 & c_0' & 0 & 0 & 0 & r_0' \\ 0 & b_1 - a_1 c_0' & c_1 & 0 & 0 & r_1 - a_0 r_0' \\ 0 & a_2 & b_2 & c_2 & 0 & r_2 \\ 0 & 0 & a_3 & b_3 & c_3 & r_3 \\ 0 & 0 & 0 & a_4 & b_4 & r_4 \end{array}\right), \tag{2.30}$$

Note that we have introduced some new symbols to simplify the notation: $c_0' = c_0/b_0$ and $r_0' = r_0/b_0$. Then we divide by $b_1 - a_1 c_0'$:

$$\left(\begin{array}{ccccc|c} 1 & c_0' & 0 & 0 & 0 & r_0' \\ 0 & 1 & c_1' & 0 & 0 & r_1' \\ 0 & a_2 & b_2 & c_2 & 0 & r_2 \\ 0 & 0 & a_3 & b_3 & c_3 & r_3 \\ 0 & 0 & 0 & a_4 & b_4 & r_4 \end{array}\right), \tag{2.31}$$

where $c_1' = c_1/(b_1 - a_1 c_0')$ and $r_1' = (r_1 - a_0 r_0')/(b_1 - a_1 c_0')$. If you continue in this manner, you can easily convince yourself that to transform a tridiagonal matrix to the following form:

$$\rightarrow \left(\begin{array}{ccccc|c} 1 & c_0' & 0 & 0 & 0 & r_0' \\ 0 & 1 & c_1' & 0 & 0 & r_1' \\ 0 & 0 & 1 & c_2' & 0 & r_2' \\ 0 & 0 & 0 & 1 & c_3' & r_3' \\ 0 & 0 & 0 & 0 & 1 & r_4' \end{array}\right), \tag{2.32}$$

where:

$$c_0' = \frac{c_0}{b_0} \qquad r_0' = r_0 b_0 \tag{2.33}$$

$$c_i' = \frac{c_i}{b_i - a_i c_{i-1}'} \qquad r_i' = \frac{r_i - a_i r_{i-1}'}{b_i - a_i c_{i-1}'} \quad \text{, for } i = 1, 2, \ldots, N-1 \tag{2.34}$$

Note that we where able to reduce the tridiagonal matrix to an *upper triangular* matrix in only *one* Gauss-Jordan step. This equation can readily be solved using back-substitution, which can also be simplified as there are a lot of zeros in the upper part. Let us denote the unknowns x_i as we did for the Gauss-Jordan case, now we can find the solution as follows:

$$x_{N-1} = r_{N-1}' \tag{2.35}$$

$$x_i = r_i' - x_{i+1} c_i' \quad \text{, for } i = N-2, N-3, \ldots, 0 \tag{2.36}$$

Equation (2.33), (2.34), (2.35) and (2.36) is known as the Thomas algorithm after Llewellyn Thomas.

Notice

Clearly tridiagonal matrices can be solved much more efficiently with the Thomas algorithm than using a standard library, such as LU-decomposition. This is because the solution method takes advantages of the *symmetry* of the problem. We will not show it here, but it can be shown that the Thomas algorithm is stable whenever $|b_i| \geq |a_i| + |c_i|$. If the algorithm fails, an advice is first to use the standard `solve` function in python. If this gives a solution, then *pivoting* combined with the Thomas algorithm might do the trick.

2.4 Example: Solving the heat equation using linear algebra

Exercise 2.1: Conservation Equation or the Continuity Equation

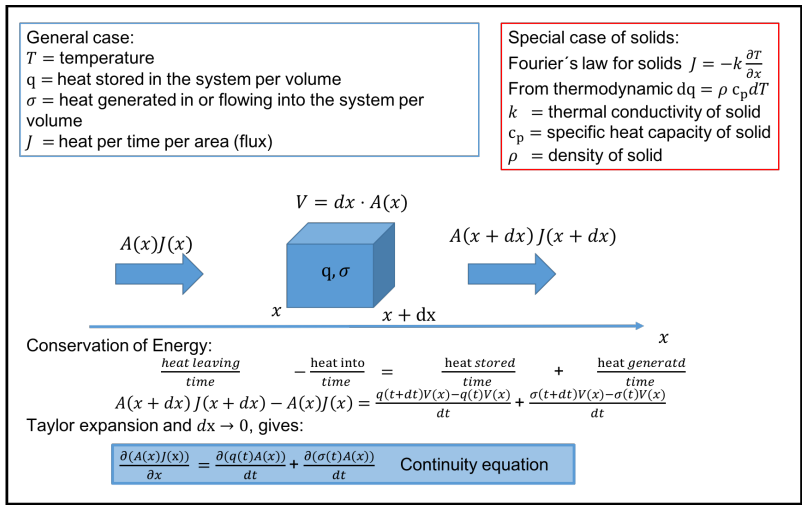

Fig. 2.2 Conservation of energy and the continuity equation.

In figure 2.2, the continuity equation is derived for heat flow. In the case of heat exchange for a solid, we can show that it can be written:

$$\frac{d^2T}{dx^2} + \frac{\dot{\sigma}}{k} = \frac{\rho c_p}{k}\frac{dT}{dt}, \tag{2.37}$$

where $\dot{\sigma}$ is the rate of heat generation in the solid. This equation can be used as a starting point for many inter-sting models. In this exercise we will investigate the *steady state* solution, *steady state* is just a fancy way of expressing that we want the solution that *does not change with time*. This is achieved by ignoring the derivative with respect to time in equation (2.37). We want to study a system with size L, and is it good practice to introduce a dimensionless variable: $y = x/L$. Equation (2.37) can now be written:

$$\frac{d^2T}{dx^2} + \frac{\dot{\sigma}L^2}{k} = 0 \tag{2.38}$$

Exercise 2.2: Curing of Concrete and Matrix Formulation

Curing of concrete is one particular example that we can investigate with equation (2.38). When concrete is curing, there are a lot of chemical reactions happening, these reactions generate heat. This is a known issue, and if the temperature rises too much compared to the surroundings, the concrete may fracture. In the following we will, for simplicity, assume that the rate of heat generated during curing is constant, $\dot{\sigma}$ =100 W/m^3. The left end (at $x = 0$) is insulated, meaning that there is no flow of heat over that boundary, hence $dT/dx = 0$ at $x = 0$. On the right hand side the temperature is kept constant, $x(L) = y(1) = T_1$, assumed to be equal to the ambient temperature of $T_1 = 25$°C. The concrete thermal conductivity is assumed to be $k = 1.65$ W/m°C.

We leave it as an exercise to show that the analytical solution to equation (2.38) in this case is:

$$T(y) = \frac{\dot{\sigma}L^2}{2k}(1 - y^2) + T_1. \tag{2.39}$$

In order to solve equation (2.38) numerically, we need to discretize it. We replace the second derivative with $dT/dy^2 = (T(y + dy) + T(y - dy) - 2T(y))/dy$. Equation (2.37) can now be written:

$$T_{i+1} + T_{i-1} - 2T_i = -h^2\beta, \tag{2.40}$$

where $\beta = 2\dot{\sigma}L^2/k$.

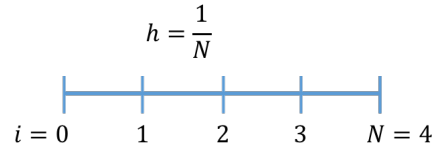

Fig. 2.3 Finite difference grid for $N = 4$.

In figure 2.3, the finite difference grid is shown for $N = 4$. Let us write down equation (2.40) for each grid node to see how the implementation is done in practice:

$$\begin{aligned} T_{-1} + T_1 - 2T_0 &= -h^2\beta, \\ T_0 + T_2 - 2T_1 &= -h^2\beta, \\ T_1 + T_3 - 2T_2 &= -h^2\beta, \\ T_2 + T_4 - 2T_3 &= -h^2\beta. \end{aligned} \tag{2.41}$$

The tricky part is now to introduce the boundary conditions. The right hand side is easy, because here the temperature is $T_4 = 25$. However, we see that T_{-1} enters and we have no value for this node. The boundary condition on the left hand side is $dT/dy = 0$, by using the central difference for the derivative allows us to write:

$$\left.\frac{dT}{dy}\right|_{y=0} = \frac{T_{-1} - T_1}{2h} = 0, \tag{2.42}$$

hence $T_{-1} = T_1$. Thus the final set of equations are:

$$\begin{aligned} 2T_1 - 2T_0 &= -h^2\beta, \\ T_0 + T_2 - 2T_1 &= -h^2\beta, \\ T_1 + T_3 - 2T_2 &= -h^2\beta, \\ T_2 + 25 - 2T_3 &= -h^2\beta, \end{aligned} \tag{2.43}$$

or in matrix form:

$$\begin{pmatrix} -2 & 2 & 0 & 0 \\ 1 & -2 & 1 & 0 \\ 0 & 1 & -2 & 1 \\ 0 & 0 & 1 & -2 \end{pmatrix} \begin{pmatrix} T_0 \\ T_1 \\ T_2 \\ T_3 \end{pmatrix} = \begin{pmatrix} -h^2\beta \\ -h^2\beta \\ -h^2\beta \\ -h^2\beta + 25 \end{pmatrix}. \tag{2.44}$$

Note that it is now easy to increase N as it is only the boundaries that requires special attention. The set of equations can be solved using `scipy.sparse.linalg.spsolve`[6]. The solution to the above equations is $L = 1$ m, and $h = 1/4$, is: $[T_0, T_1.T_2, T_3] = [38.88888889, 38.02083333, 35.41666667, 31.07638889]$.

Solution.

[6] `https://docs.scipy.org/doc/scipy/reference/sparse.linalg.html`

Notice

The solution below implements equation (2.2) using sparse matrices, and the standard Numpy `solve` function. You can use the `%timeit` magic command in Ipython and Jupyter notebooks to test the efficiency.

```
#%matplotlib inline
import numpy as np
import scipy as sc
import scipy.sparse.linalg
from numpy.linalg import solve
import matplotlib.pyplot as plt

# Set simulation parameters
h = 0.25               # element size
L = 1.0               # length of domain
n = int(round(L/h))   # number of unknowns
x=np.arange(n+1)*h    # includes right bc
T1=25
sigma = 100*L**2/1.65

def tri_diag(a, b, c, k1=-1, k2=0, k3=1):
    """ a,b,c diagonal terms """
    return np.diag(a, k1) + np.diag(b, k2) + np.diag(c, k3)

def analytical(sigma,x):
    return sigma*(1-x*x)/2+T1

#Create matrix for linalg solver
a=np.ones(n-1)
b=-np.ones(n)*2
c=np.ones(n-1)
#Create matrix for sparse solver
diagonals=np.zeros((3,n))
diagonals[0,:]= 1
diagonals[1,:]= -2
diagonals[2,:]= 1

# rhs vector
d=np.repeat(-h*h*sigma,n)

#----boundary conditions ------
#lhs - no flux of heat
diagonals[2,1]= 2
c[0]=2
#rhs - constant temperature
d[n-1]=d[n-1]-T1
#------------------------------
```

```
A=tri_diag(a,b,c)
A_sparse = sc.sparse.spdiags(diagonals, [-1,0,1], n, n,format='csc')

#Solve linear problems
Ta = solve(A,d,)
Tb = sc.sparse.linalg.spsolve(A_sparse,d)
#Add right boundary node
Ta=np.append(Ta,T1)
Tb=np.append(Tb,T1)
#uncomment to test efficiency
#%timeit sc.sparse.linalg.spsolve(A_sparse,d)
#%timeit solve(A,d,)

# Plot solutions
plt.plot(x,Ta,x,Tb,'-.',x,analytical(sigma,x),':', lw=3)
plt.xlabel("Dimensionless length")
plt.ylabel(r"Temperature [$^\circ$C]")
plt.xlim(0,1)
plt.ylim(T1-1)
plt.legend(['sparse','linalg','analytical'])
plt.grid()
plt.show()
```

Numerical integration 3

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 $k = 0$.

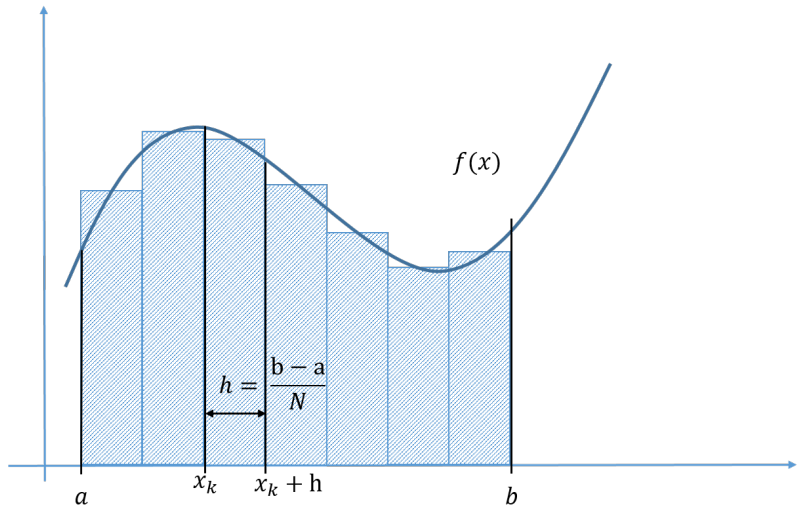

Fig. 3.1 Integrating a function with the midpoint rule.

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(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 = 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
```

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.

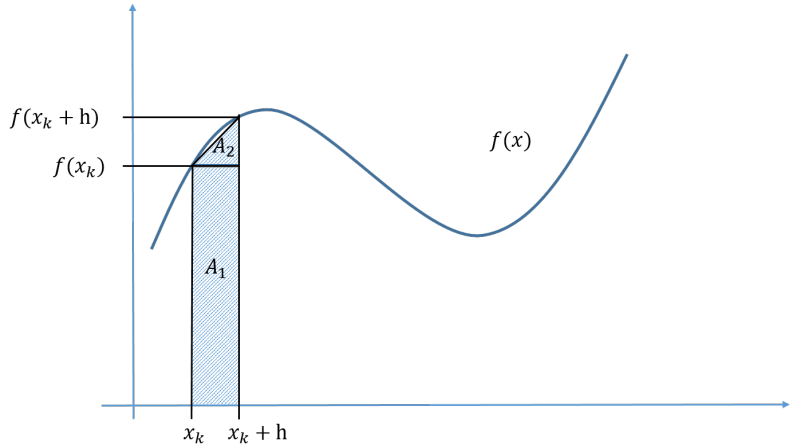

Fig. 3.2 Integrating a function with the trapezoidal rule.

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:

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

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:

```
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 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 $f(x)$:

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

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

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

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:

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

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

The corresponding formula for the trapezoid formula is:

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

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(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}.

3.1.4 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:

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

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:

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

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

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:

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

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

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

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.

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 use these two estimates to calculate the error term:

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

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

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.

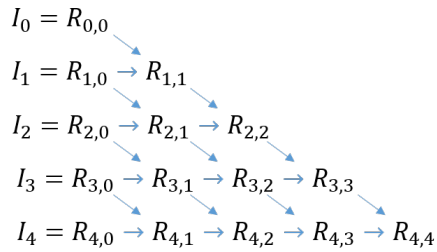

Fig. 3.3 Illustration of the Romberg algorithm. Note that for each new evaluation of the integral $R_{i,0}$, all the correction terms $R_{i,m}$ (for $m > 0$) must be evaluated again.

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:

```
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 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

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

from scipy import integrate
integrate.romberg(f, a, b, show=True)

def g(x):
    u=(1-x)
    u*=u
    return np.exp(-x*x/u)/u

w=100
def h(x):
    u=x/(1-x)
    return u*np.exp(-u)*np.cos(w*u)/(1-x)/(1-x)

Area = int_romberg(g,0.,0.99999999,prec,show=True)
print('Numerical value = ', Area)
print('Error           = ', (np.sqrt(np.pi)/2-Area)) # Analytical result is 2

Area = int_romberg(h,0.,0.99999999,prec,show=True)
print('Numerical value = ', Area)
```

```
print('Error           = ', ((1-w*w)/(w**2+1)**2-Area)) # Analytical result is 2

integrate.romberg(h, 0, 0.99999999999, show=True)

def i(x):
    return x**4-2*x+1

int_romberg(i,0.,2,prec,show=True)
```

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:

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

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

```
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:

```
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 $11^2 = 121$.

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 $E_T = -2E_M$.

$$I(x_k, x_k + h) = M(x_k, x_k + h) + E_M \text{ (midpoint rule)} \quad (3.55)$$

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

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:

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

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.

Solving nonlinear equations

4

Ordinary differential equations 5

5.1 Ordinary Differential Equations

Physical systems evolves in space and time, and very often they are described by a ordinary differential equations (ODE) and/or partial differential equations (PDE). The difference between an ODE and a PDE is that an ODE only describes the changes in one spatial dimension *or* time, whereas a PDE describes a system that evolves in the $x-$, $y-$, $z-$ dimension and/or in time. In the following we will spend a significant amount of time to explore one of the simplest algorithm, Eulers method. Sometimes this is exactly the algorithm you would like to use, but with very little extra effort much more sophisticated algorithms can easily be implemented, such as the Runge-Kutta fourth order method. However, all these algorithms, will at some point run into the same kind of troubles if used reckless. Thus we will use the Eulers method as a playground, investigate when the algorithm run into trouble and suggests ways to fix it, these approaches can easily be extended to the higher order methods. Most of the other algorithms boils down to the same idea of extrapolating a function using derivatives multiplied with a small step size.

5.2 A Simple Model for Fluid Flow

Let us consider a simple example from chemical engineering, a continuous stirred tank reactor (CSTR), see figure 5.1. The flow is incompressible

($q_{\text{out}} = q_{\text{in}}$), a fluid is entering on the top and exiting at the bottom, the tank has a fixed volume V. Assume that the tank is filled with saltwater, and that freshwater is pumped into it, how much time does it take before 90% of the saltwater is replaced with freshwater? The tank is *well mixed*, illustrated with the propeller, this means that at every time the concentration is uniform in the tank, i.e. that $C(t) = C_{\text{out}}(t)$.

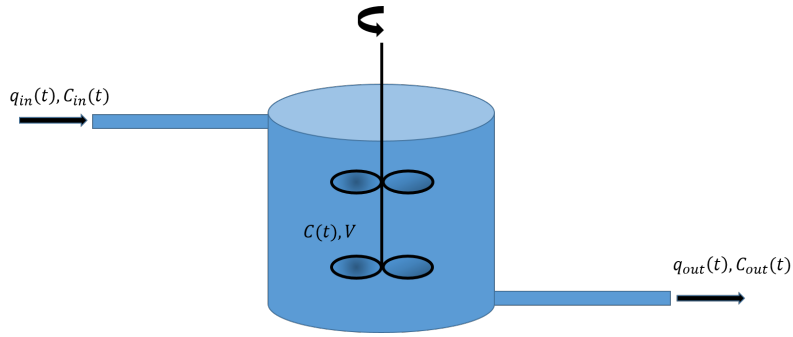

Fig. 5.1 A continuous stirred tank model, $C(t) = C_{\text{out}}(t)$, and $q_{\text{out}} = q_{\text{in}}$.

The concentration C is measured in gram of salt per liter water, and the flow rate q is liter of water per day. The model for the salt balance in this system can be described in words by:

$$\begin{aligned}[\text{accumulation of salt}] = [\text{salt into the system}] - [\text{salt out of the system}] \\ + [\text{generation of salt}]. \end{aligned} \tag{5.1}$$

In our case there are no generation of salt within the system so this term is zero. The flow of salt into the system during a time Δt is: $q_{\text{in}}(t) \cdot C_{\text{in}}(t) \cdot \Delta t = q(t) \cdot C_{\text{in}}(t) \cdot \Delta t$, the flow of salt out of the system is: $q_{\text{out}}(t) \cdot C_{\text{out}}(t) \cdot \Delta t = q(t) \cdot C(t) \cdot \Delta t$, and the accumulation during a time step is: $C(t + \Delta t) \cdot V - C(t) \cdot V$, hence:

$$C(t + \Delta t) \cdot V - C(t) \cdot V = q(t) \cdot C_{\text{in}}(t) \cdot \Delta t - q(t) \cdot C(t) \cdot \Delta t. \tag{5.2}$$

Note that it is not a priori apparent, which time the concentrations and flow rates on the right hand side should be evaluated at, we could have chosen to evaluate them at $t + \Delta t$, or at any time $t \in [t, t + \Delta t]$. We will return to this point later in this chapter. Dividing by Δt, and taking the limit $\Delta t \to 0$, we can write equation (5.2) as:

$$V \frac{dC(t)}{dt} = q(t) \left[C_{\text{in}}(t) - C(t) \right]. \tag{5.3}$$

Seawater contains about 35 gram salt/liter fluid, if we assume that the fresh water contains no salt, we have the boundary conditions $C_{\text{in}}(t) = 0$, $C(0)$ =35gram/l. The equation (5.3) the reduces to:

$$V\frac{dC(t)}{dt} = -qC(t), \tag{5.4}$$

this equation can easily be solved, by dividing by C, multiplying by dt and integrating:

$$V\int_{C_0}^{C}\frac{dC}{C} = -q\int_0^t dt,$$
$$C(t) = C_0 e^{-t/\tau}, \text{ where } \tau \equiv \frac{V}{q}. \tag{5.5}$$

This equation can be inverted to give $t = -\tau \ln[C(t)/C]$. If we assume that the volume of the tank is 1m^3=1000liters, and that the flow rate is 1 liter/min, we find that τ=1000min=0.69days and that it takes about $-0.69 \ln 0.9 \simeq 1.6$days to reduce the concentration by 90% to 3.5 gram/liter.

The CSTR

You might think that the CSTR is a very simple model, and it is, but this type of model is the basic building blocks in chemical engineering. By putting CSTR tanks in series and/or connecting them with pipes, the efficiency of manufacturing various type of chemicals can be investigated. Although the CSTR is an idealized model for the part of a chemical factory, it is actually a *very good* model for fluid flow in a porous media. By connecting CSTR tanks in series, one can model how chemical tracers propagate in the subsurface. The physical reason for this is that dispersion in porous media will play the role of the propellers and mix the concentration uniformly.

5.3 Eulers Method

If the system gets slightly more complicated, e.g several tanks in series with a varying flow rate or if salt was generated in the tank, there is a good chance that we have to solve the equations numerically to obtain a

solution. Actually, we have already developed a numerical algorithm to solve equation (5.3), before we arrived at equation (5.3) in equation (5.2). This is a special case of Eulers method, which is basically to replace the derivative in equation (5.3), with $(C(t+\Delta t)-C(t))/\Delta t$. By rewriting equation (5.2), so that we keep everything related to the new time step, $t+\Delta t$, on one side, we get:

$$VC(t+\Delta t) = VC(t) + qC_{\text{in}}(t) - qC(t), \tag{5.6}$$

$$C(t+\Delta t) = C(t) + \frac{\Delta t}{\tau}\left[C_{\text{in}}(t) - C(t)\right], \tag{5.7}$$

we introduce the short hand notation: $C(t) = C_n$, and $C(t+\Delta t) = C_{n+1}$, hence the algorithm can be written more compact as:

$$C_{n+1} = \left(1 - \frac{\Delta t}{\tau}\right) C_n + \frac{\Delta t}{\tau} C_{\text{in},n}, \tag{5.8}$$

In the script below, we have implemented equation (5.8).

```
def analytical(x):
    return np.exp(-x)

def euler_step(c_old, c_in, tau_inv,dt):
    fact=dt*tau_inv
    return (1-fact)*c_old+fact*c_in

def ode_solv(c_into,c_init,t_final,vol,q,dt):
    f=[];t=[]
    tau_inv = q/vol
    c_in    = c_into #freshwater into tank
    c_old   = c_init #seawater present
    ti=0.
    while(ti <= t_final):
        t.append(ti); f.append(c_old)
        c_new = euler_step(c_old,c_in,tau_inv,dt)
        c_old = c_new
        ti   += dt
    return t,f
```

In figure 5.2 the result of the implementation is shown for different values of Δt Clearly we see that the results are dependent on the step size, as the step increases the numerical solution deviates from the analytical solution. At some point the numerical algorithm fails completely, and produces results that have no meaning.

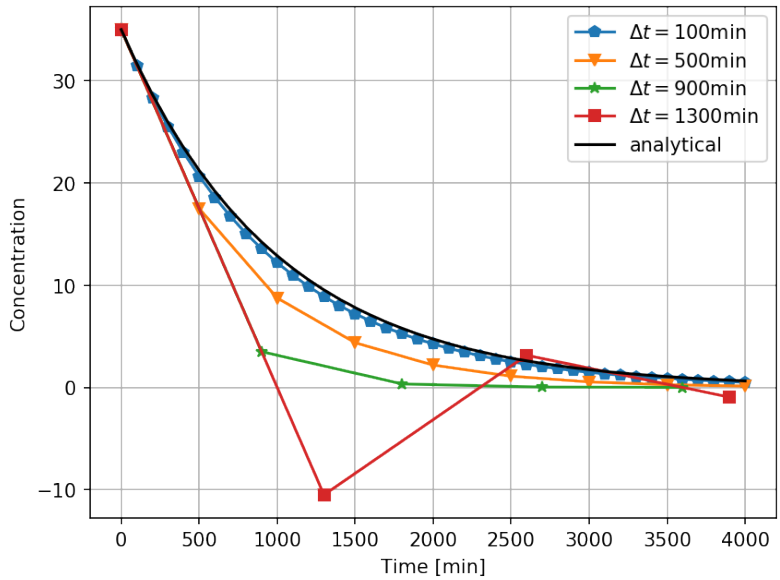

Fig. 5.2 The concentration in the tank for different step size Δt.

5.3.1 Error Analysis - Eulers Method

There are two obvious questions:

1. When does the algorithm produce unphysical results?
2. What is an appropriate step size?

Let us consider the first question, clearly when the concentrations gets negative the solution is unphysical. From equation (5.8), we see that when $\Delta t/\tau > 1$, the concentration become negative. For this specific case (the CSTR), there is a clear physical interpretation of this condition. Inserting $\tau = V/q$, we can rewrite the condition $\Delta t/\tau < 1$ as $q\Delta t < V$. The volume into the tank during one time step is: $q\Delta t$, which means that whenever we *flush more than one tank volume through the tank during one time step, the algorithm fails*. When this happens the new concentration in the tank cannot be predicted from the old one. This makes sense, because we could have switched to a new solution (e.g. seawater) during that time step, then the new solution does not have any relation to the old solution.

The second question, "what is an appropriate step size?", is a bit more difficult to answer. One strategy could be to simply use the results from chapter [Taylor], where we showed that the truncation error had

a minimum value with a step size of 10^{-8} (when using a first order Taylor approximation). How does the value 10^{-8} relate to the step sizes in minutes used in our Euler implementation? In order to see the connection, we need to rewrite equation (5.3) in a dimensionless form, by making the following substitution: $t \rightarrow t/\tau$:

$$\frac{dC(\tau)}{d\tau} = [C_{\text{in}}(\tau) - C(\tau)]\,. \tag{5.9}$$

As we found earlier $\tau = 1000$min, thus a step size of e.g. 1 min would correspond to a dimensionless time step of $\Delta t \rightarrow$1min/1000min$= 10^{-3}$. This number can be directly compared to the value 10^{-8}, which is the lowest value we can choose without getting into trouble with round off errors on the machine.

Dimensionless variables

It is a good idea (necessary) to formulate our equations in terms of dimensionless variables. The algorithms we develop can then be used in the same form regardless of changes in the system size and flow rates. Thus we do not need to rewrite the algorithm each time the physical system changes. This also means that if you use an algorithm developed by someone else (e.g. in Matlab or Python), you should always formulate the ODE system in dimensionless form before using the algorithm.

A second reason is that from a pure modeling point of view, dimensionless variables is a way of getting some understanding of what kind of combination of the physical parameters that describes the behavior of the system. For the case of the CSTR, there is a time scale $\tau = V/q$, which is an intrinsic measure of time in the system. No matter what the flow rate through the tank or the volume of the tank is, it will always take 0.1τ before the concentration in the tank is reduced by 90%.

As already mentioned a step size of 10^{-8}, is probably the smallest we can choose with respect to round off errors, but it is smaller than necessary and would lead to large simulation times. If it takes 1 second to run the simulation with a step size of 10^{-3}, it would take 10^5 seconds or 1 day with a step size of 10^{-8}. To continue the error analyses, we write our ODE for a general system as:

$$\frac{dy}{dt} = f(y,t), \tag{5.10}$$

or in discrete form:

$$\frac{y_{n+1} - y_n}{h} - \frac{h}{2}y''(\eta_n) = f(y,t).$$

$$y_{n+1} = y_n + hf(y,t) + \frac{h^2}{2}y''(\eta_n). \tag{5.11}$$

h is now the (dimensionless) step size, equal to Δt if the derivative is with respect to t or Δx if the derivative is respect to x etc. Note that we have also included the error term related to the numerical derivative, $\eta_n \in [t_n, t_n + h]$. At each step we get an error term, and the distance between the true solution and our estimate, the *local error*, after N steps is:

$$\epsilon = \sum_{n=0}^{N-1} \frac{h^2}{2}y''(\eta_n) = \frac{h^2}{2}\sum_{n=0}^{N-1} f'(y_n, \eta_n) \simeq \frac{h}{2}\int_{t_0}^{t_f} f'(y,\eta)d\eta$$

$$= \frac{h}{2}\left[f(y(t_f), t_f) - f(y(t_0), t_0)\right]. \tag{5.12}$$

Note that when we replace the sum with an integral in the equation above, this is only correct if the step size is not too large. From equation (5.12) we see that even if the error term on the numerical derivative is h^2, the local error is proportional to h (one order lower). This is because we accumulate errors for each step.

In the following we specialize to the CSTR, to see if we can gain some additional insight. First we change variables in equation (5.4): $y = C(t)/C_0$, and $x = t/\tau$, hence:

$$\frac{dy}{dx} = -y. \tag{5.13}$$

The solution to this equation is $y(x) = e^{-x}$, substituting back for the new variables y and x, we reproduce the result in equation (5.5). The local error, equation (5.12), reduces to:

$$\epsilon = \frac{h}{2}\left[-y(x_f) + y(x_0)\right] = \frac{h}{2}\left[1 - e^{-x_f}\right], \tag{5.14}$$

we have assumed that $x_0 = t_0/\tau = 0$. This gives the estimated local error at time x_f. For $x_f = 0$, the numerical error is zero, this makes sense because at $x = 0$ we know the exact solution because of the initial

conditions. When we move further away from the initial conditions, the numerical error increases, but equation (5.14) ensures us that as long as the step size is low enough we can get as close as possible to the true solution, since the error scales as h (at some point we might run into trouble with round off error in the computer).

Can we prove directly that we get the analytical result? In this case it is fairly simple, if we use Eulers method on equation (5.13), we get:

$$\frac{y_{n+1} - y_n}{h} = -y_n f.$$
$$y_{n+1} = (1-h)y_n, \tag{5.15}$$

or alternatively:

$$\begin{aligned} y_1 &= (1-h)y_0, \\ y_2 &= (1-h)y_1 = (1-h)^2 y_0, \\ &\vdots \\ y_{N+1} &= (1-h)^N y_0 = (1-h)^{x_f/h} y_0. \end{aligned} \tag{5.16}$$

In the last equation, we have used the the fact the number of steps, N, is equal to the simulation time divided by the step size, hence: $N = x_f/h$. From calculus, the equation above is one of the well known limits for the exponential function: $\lim_{x\to\infty}(1 + k/x)^{mx} = e^{mk}$, hence:

$$y_n = (1-h)^{x_f/h} y_0 \to e^{-x_f}, \tag{5.17}$$

when $h \to 0$. Below is an implementation of the Euler algorithm in this simple case, we also estimate the local error, and global error after N steps.

```
import matplotlib.pyplot as plt
import numpy as np
def euler(tf,h):
    t=[];f=[]
    ti=0.;fi=1.
    t.append(ti);f.append(fi)
    global_err=0.
    while(ti<= tf):
        ti+=h
        fi=fi*(1-h)
        global_err += abs(np.exp(-ti)-fi)
        t.append(ti);f.append(fi)
    print("error= ", np.exp(-ti)-fi," est.err=", .5*h*(1-np.exp(-ti)))
    print("global error=",global_err)
```

```
    return t,f

t,f=euler(1,1e-5)
```

By changing the step size h, you can easily verify that the local error systematically increases or decreases proportional to h. Something curious happens with the global error when the step size is changed, it does not change very much. The global error involves a second sum over the local error for each step, which can be approximated as a second integration in equation (5.14):

$$\epsilon_{\text{global}} = \frac{1}{2}\int_0^{x_f} \left[-y(x) + y(0)\right] dx = \frac{1}{2}\left[x_f + e^{-x_f} - 1\right]. \tag{5.18}$$

Note that the global error does not go to zero when the step size decreases, which can easily be verified by changing the step size. This is strange, but can be understood by the following argument: when the step size decreases the local error scales as $\sim h$, but the number of steps scales as $1/h$, so the global error must scale as $h \times 1/h$ or some constant value. Usually it is much easier to control the local error than the global error, this should be kept in mind if you ever encounter a problem where it is important control the global error. For the higher order methods that we will discuss later in this chapter, the global error will go to zero when h decreases.

The answer to our original question, "What is an appropriate step size?", will depend on what you want to achieve in terms of local or global error. In most practical situations you would specify a local error that is acceptable for the problem under investigation and then choose a step size where the local error always is lower than this value. In the next subsection we will investigate how to achieve this in practice.

5.3.2 Adaptive step size - Eulers Method

We want to be sure that we use a step size that achieves a certain accuracy in our numerical solution, but at the same time that we do not waste simulation time using a too low step size. The following approach is similar to the one we derived for the Romberg integration, and a special case of what is known as Richardson Extrapolation. The method is easily extended to higher order methods.

We know that Eulers algorithm is accurate to second order. Our estimate of the new value, y_1^* (where we have used a * to indicate that

we have used a step size of size h), should then be related to the true solution $y(t_1)$ in the following way:

$$y_1^* = y(t_1) + ch^2. \tag{5.19}$$

The constant c is unknown, but it can be found by taking two smaller steps of size $h/2$. If the steps are not too large, our new estimate of the value y_1 will be related to the true solution as:

$$y_1 = y(t_1) + 2c\left(\frac{h}{2}\right)^2. \tag{5.20}$$

The factor 2 in front of c is because we now need to take two steps, and we accumulate a total error of $2c(h/2)^2 = ch^2/2$. It might not be completely obvious that the constant c should be the same in equation (5.19) and (5.20). If you are not convinced, there is an exercise at the end of the chapter. We define:

$$\Delta \equiv y_1^* - y_1 = c\frac{h^2}{2}. \tag{5.21}$$

The truncation error in equation (5.20) is:

$$\epsilon = y(t_1) - y_1 = 2c\left(\frac{h}{2}\right)^2 = \Delta. \tag{5.22}$$

Now we have everything we need: We want the local error to be smaller than some predefined tolerance, ϵ', or equivalently that $\epsilon \leq \epsilon'$. To achieve this we need to use an optimal step size, h', that gives us exactly the desired error:

$$\epsilon' = c\frac{h'^2}{2}. \tag{5.23}$$

Dividing equation (5.23) by equation (5.22), we can estimate the optimal step size:

$$h' = h\sqrt{\left|\frac{\epsilon'}{\epsilon}\right|}, \tag{5.24}$$

where the estimated error, ϵ, is calculated from equation (5.22). Equation (5.24) serves two purposes, if the estimated error ϵ is higher than the tolerance, ϵ', we have specified it will give us an estimate for the step size we should choose in order to achieve a higher accuracy, if on the other

hand $\epsilon' > \epsilon$, then we get an estimate for the next, larger step. Before the implementation we note, as we did for the Romberg integration, that equation (5.22) also gives us an estimate for the error term in equation (5.20) as an improved estimate of y_1. This we get for free and will make our Euler algorithm accurate to h^3, hence the improved Euler step, $\hat{y_1}$, is to *subtract* the error term from our previous estimate:

$$\hat{y_1} = y_1 - \epsilon = 2y_1 - y_1^*. \tag{5.25}$$

Below is an implementation of the adaptive Euler algorithm:

```
def one_step(c_old, c_in,h):
    return (1-h)*c_old+h*c_in

def adaptive_euler(c_into,c_init,t_final,tol=1e-4):
    f=[];t=[]
    c_in    = c_into #freshwater into tank
    c_old   = c_init #seawater present
    ti=0.; h_new=1e-3;
    toli=1.; # a high init tolerance to enter while loop
    no_steps=0
    global_err=0.
    while(ti <= t_final):
        t.append(ti); f.append(c_old)
        while(toli>tol):# first two small steps
            hi=h_new
            k1 = one_step(c_old,c_in,hi*.5)
            k2 = one_step(k1,c_in,hi*.5)
            # ... and one large step
            k3 = one_step(c_old,c_in,hi)
            toli = abs(k3-k2)
            h_new=hi*np.sqrt(tol/toli)
            no_steps+=3
        toli=1.
        c_old=2*k2-k3 # higher order correction
 # normal Euler, uncomment and inspect the global error
 #       c_old = k2
        ti   += hi
        global_err += abs(np.exp(-ti)-c_old)
    print("No steps=", no_steps, "Global Error=", global_err)
    return t,f
```

In figure 5.3 the result of the implementation is shown. Note that the number of steps for an accuracy of 10^{-6} is only about 3000. Without knowing anything about the accuracy, we would have to assume that we needed a step size of the order of h in order to reach a local accuracy of h because of equation (5.12). In the current case, we would have needed 10^7 steps, which would lead to unnecessary long simulation times.

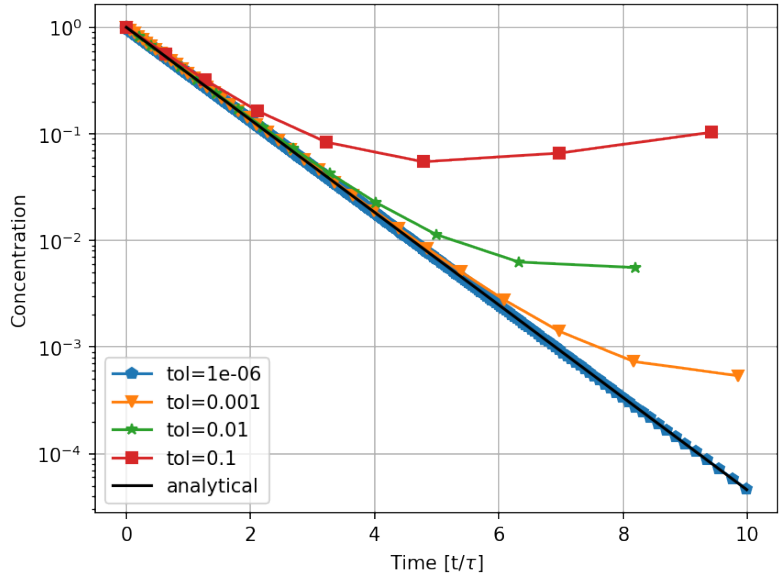

Fig. 5.3 The concentration in the tank using adaptive Euler. Number of Euler steps are: 3006, 117, 48 and 36 for the different step sizes.

Local error and bounds

In the previous example we set an absolute tolerance, and required that our estimate y_n always is within a certain bound of the true solution $y(t_n)$, i.e. $|y(t_n) - y_n| \leq \epsilon'$. This is a very strong demand, and sometimes it makes more sense to require that we also accept a relative tolerance proportional to function value. In some areas the solution might have a very large value, and then another possibility would be to have an ϵ' that varied with the function value: $\epsilon' = atol + |y|rtol$, where 'atol' is the absolute tolerance and 'rtol' is the relative tolerance. A sensible choice would be to set 'atol=rtol' (e.g. $= 10^{-4}$).

5.4 Runge-Kutta Methods

The Euler method only have an accuracy of order h, and a global error that do not go to zero as the step size decrease. The Runge-Kutta methods may be motivated by inspecting the Euler method in figure 5.4.

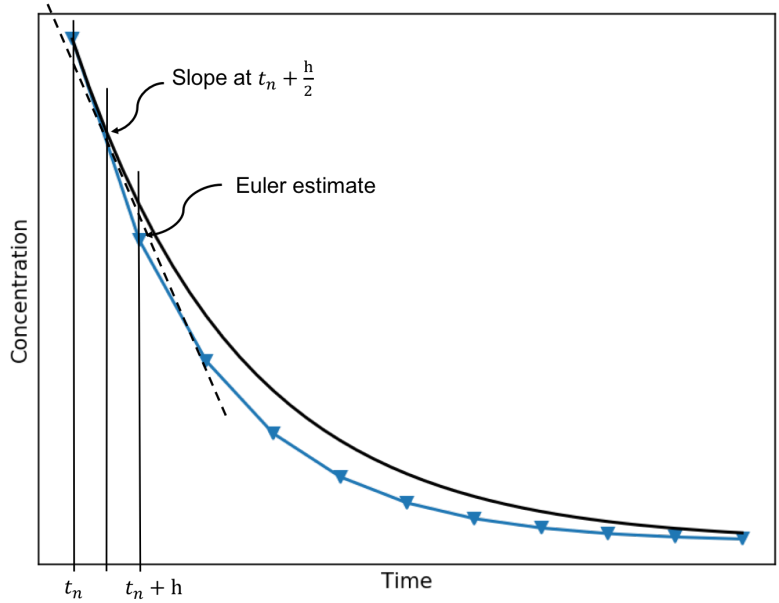

Fig. 5.4 Illustration of the Euler algorithm, and a motivation for using the slope a distance from the t_n.

The Euler method uses information from the previous time step to estimate the value at the new time step. The Runge Kutta methods uses the information about the slope between the points t_n and $t_n + h$. By inspecting figure 5.4, we clearly see that by using the slope at $t_n + h/2$ would give us a significant improvement. The 2. order Runge-Kutta method can be derived by Taylor expanding the solution around $t_n + h/2$, we do this by setting $t_n + h = t_n + h/2 + h/2$:

$$y(t_n + h) = y(t_n + \frac{h}{2}) + \frac{h}{2}\frac{dy}{dt}\bigg|_{t=t_n+h/2} + \frac{h^2}{4}\frac{d^2y}{dt^2}\bigg|_{t=t_n+h/2} + \mathcal{O}(h^3). \tag{5.26}$$

Similarly we can expand the solution in $y(t_n)$ about $t_n + h/2$, by setting $t_n = t_n + h/2 - h/2$:

$$y(t_n) = y(t_n + \frac{h}{2}) - \frac{h}{2}\frac{dy}{dt}\bigg|_{t=t_n+h/2} + \frac{h^2}{4}\frac{d^2y}{dt^2}\bigg|_{t=t_n+h/2} - \mathcal{O}(h^3). \tag{5.27}$$

Subtracting these two equations the term $y(t_n + \frac{h}{2})$, and all even powers in the derivative cancels out:

$$y(t_n + h) = y(t_n) + h \left.\frac{dy}{dt}\right|_{t=t_n+h/2} + \mathcal{O}(h^3),$$
$$y(t_n + h) = y(t_n) + hf(y_{n+h/2}, t_n + h/2) + \mathcal{O}(h^3). \tag{5.28}$$

In the last equation, we have used equation (5.10). Note that we now have an expression that is very similar to Eulers algorithm, but it is accurate to order h^3. There is one problem, and that is that the function f is to be evaluated at the point $y_{n+1/2} = y(t_n + h/2)$ which we do not know. This can be fixed by using Eulers algorithm: $y_{n+1/2} = y_n + h/2f(y_n, t_n)$. We can do this even if Eulers algorithm has an error term of order h^2, because the f in equation (5.28) is multiplied by h, and thus our algorithm is still has an error term of order h^3.

The 2. order Runge-Kutta:

$$\begin{aligned} k_1 &= hf(y_n, t_n) \\ k_2 &= hf(y_n + \frac{1}{2}k_1, t_n + h/2) \\ y_{n+1} &= y_n + k_2 \end{aligned} \tag{5.29}$$

Below is a Python implementation of equation (5.29):

```
def fm(c_old,c_in):
    return c_in-c_old

def rk2_step(c_old, c_in, h):
    k1=h*fm(c_old,c_in)
    k2=h*fm(c_old+0.5*k1,c_in)
    return c_old+k2

def ode_solv(c_into,c_init,t_final,h):
    f=[];t=[]
    c_in  = c_into #freshwater into tank
    c_old = c_init #seawater present
    ti=0.
    while(ti <= t_final):
        t.append(ti); f.append(c_old)
        c_new = rk2_step(c_old,c_in,h)
        c_old = c_new
        ti   += h
    return t,f
```

In figure 5.5 the result of the implementation is shown. Note that when comparing Runge-Kutta 2. order with Eulers method, see figure 5.5

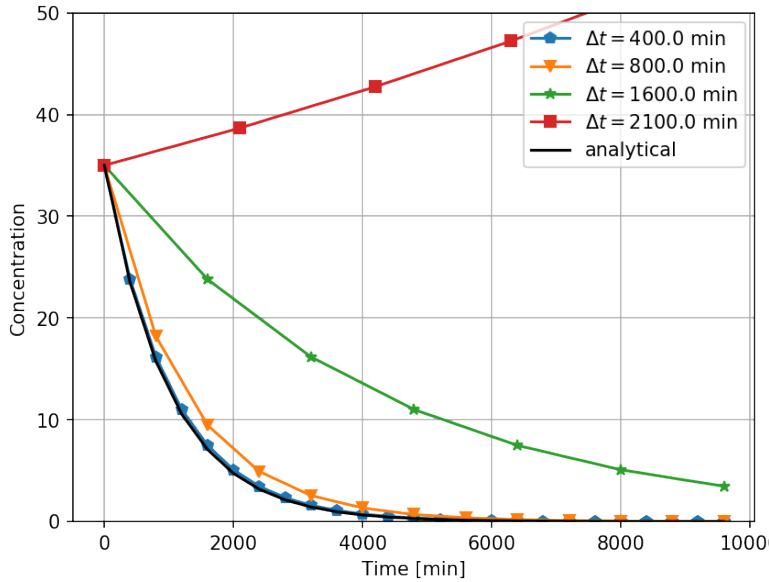

Fig. 5.5 The concentration in the tank for different step size Δt.

and 5.2, we of course have the obvious result that a larger step size can be taken, without loosing numerical accuracy. It is also worth noting that we can take steps that is larger than the tank volume. Eulers method failed whenever the time step was larger than one tank volume ($h = t/\tau > 1$), whereas the Runge-Kutta method finds a physical solution for step sizes lower than twice the tank volume. If the step size is larger, we see that the concentration in the tank increases, which is clearly unphysical.

The Runge-Kutta fourth order method is one of he most used methods, it is accurate to order h^4, and has an error of order h^5. The development of the algorithm itself is similar to the 2. order method, but of course more involved. We just quote the result:

The 4. order Runge-Kutta:

$$
\begin{aligned}
k_1 &= hf(y_n, t_n) \\
k_2 &= hf(y_n + \frac{1}{2}k_1, t_n + h/2) \\
k_3 &= hf(y_n + \frac{1}{2}k_2, t_n + h/2) \\
k_4 &= hf(y_n + k_3, t_n + h) \\
y_{n+1} &= y_n + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4) \qquad (5.30)
\end{aligned}
$$

In figure 5.6 the result of the Runge-Kutta fourth order is shown, by comparing it to figure 5.5 it is easy to see that a larger step size can be chosen.

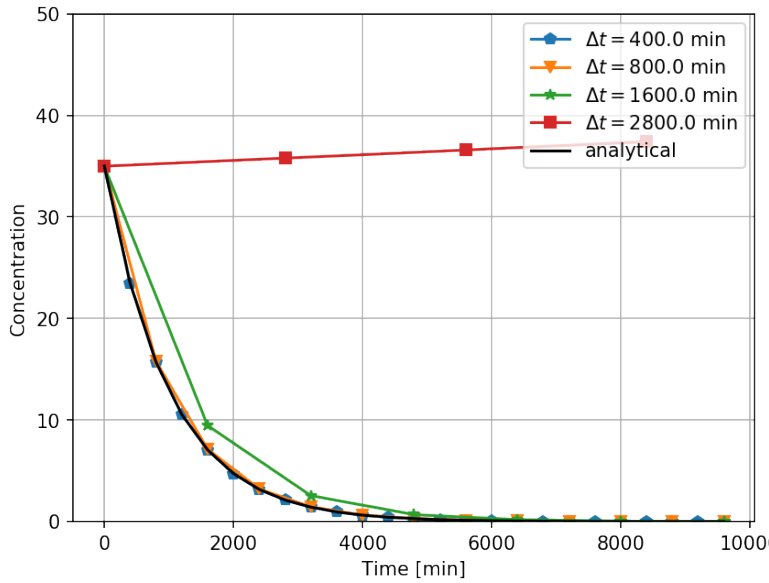

Fig. 5.6 The concentration in the tank for different step size Δt.

5.4.1 Adaptive step size - Runge-Kutta Method

Just as we did with Eulers method, we can implement an adaptive method. The derivation is exactly the same, but this time our method is accurate to fourth order, hence the error term is of order h^5. We start

by taking one large step of size h, our estimate, y_1^* is related to the true solution, $y(t_1)$, in the following way:

$$y_1^* = y(t_1) + ch^5, \tag{5.31}$$

Next, we take two steps of half the size, $h/2$, hence:

$$y_1 = y(t) + 2c\left(\frac{h}{2}\right)^5. \tag{5.32}$$

Subtracting equation (5.31) and (5.32), we find an expression similar to equation (5.21):

$$\Delta \equiv y_1^* - y_1 = c\frac{15}{16}h^5, \tag{5.33}$$

or $c = 16\Delta/(15h^5)$. For the Euler scheme, Δ also happened to be equal to the truncation error, but in this case it is:

$$\epsilon = 2c\left(\frac{h}{2}\right)^5 = \frac{\Delta}{15} \tag{5.34}$$

we want the local error, ϵ, to be smaller than some tolerance, ϵ'. The optimal step size, h', that gives us exactly the desired error is then:

$$\epsilon' = 2c\left(\frac{h'}{2}\right)^5. \tag{5.35}$$

Dividing equation (5.35) by equation (5.34), we can estimate the optimal step size:

$$h' = h\left|\frac{\epsilon}{\epsilon}\right|^{1/5}, \tag{5.36}$$

ϵ can be calculated from equation (5.34). In figure 5.7 the result of an implementation is shown (see the exercises).

In general we can use the same procedure any method accurate to order h^p, and you can easily verify that:

Error term and step size for a h^p method:

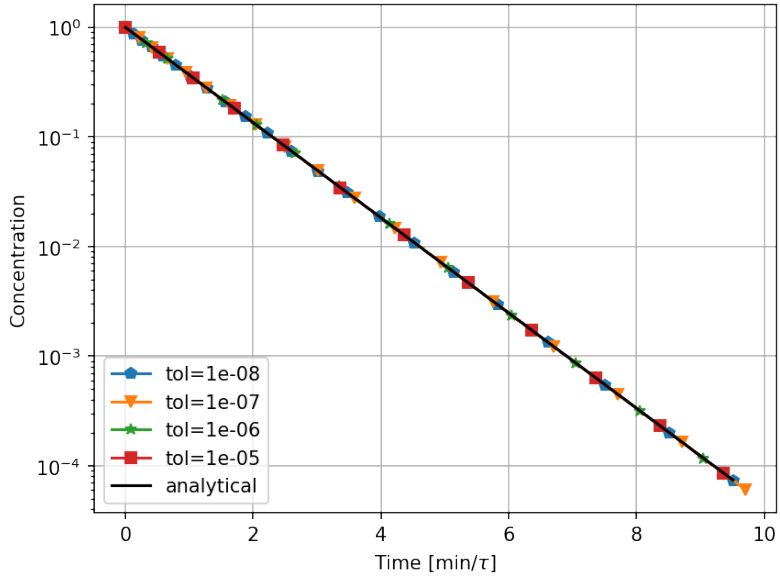

Fig. 5.7 The concentration in the tank for different step size Δt. Number of rk4 steps are: 138, 99, 72 and 66 for the different step sizes and 'rtol=0', for 'rtol=tol' the number of rk4 steps are 81, 72, 63, 63.

$$|\epsilon| = \frac{|\Delta|}{2^p - 1} = \frac{|y_1^* - y_1|}{2^p - 1}, \tag{5.37}$$

$$h' = \beta h \left| \frac{\epsilon}{\epsilon_0} \right|^{\frac{1}{p+1}}, \tag{5.38}$$

$$\hat{y_1} = y_1 - \epsilon = \frac{2^p y_1 - y_1^*}{2^p - 1}, \tag{5.39}$$

where β is a safety factor $\beta \simeq 0.8, 0.9$, and you should always be careful that the step size do not become too large so that the method breaks down. This can happens when ϵ is very low, which may happen if $y_1^* \simeq y_1$ and/or if $y_1^* \simeq y_1 \simeq 0$.

5.4.2 Conservation of Mass

A mathematical model of a physical system should always be formulated in such a way that it is consistent with the laws of nature. In practical situations this statement is usually equivalent to state that the mathematical model should respect conservation laws. The conservation laws can be conservation of mass, energy, momentum, electrical charge,

etc. In our example with the mixing tank, we were able to derive an expression for the concentration of salt out of the tank, equation (5.5), by *demanding* conservation of mass (see equation (5.2)).

A natural question to ask is then: If our mathematical model respect conservation of mass, are we sure that our solution method respect conservation of mass? We of course expect that when the grid spacing approaches zero our numerical solution will get closer and closer to the analytical solution. Clearly when $\Delta x \to 0$, the mass is conserved. So what is the problem? The problem is that in many practical problems we cannot always have a step size that is small enough to ensure that our solution always is close enough to the analytical solution. The physical system we consider might be very complicated (e.g. a model for the earth climate), and our ODE system could be a very small part of a very big system. A very good test of any code is to investigate if the code respect the conservation laws. If we know that our implementation respect e.g. mass conservation at the discrete level, we can easily test mass conservation by summing up all the mass entering, and subtracting the mass out of and present in our system. If the mass is not conserved exactly, there is a good chance that there is a bug in our implementation.

If we now turn to our system, we know that the total amount of salt in the system when we start is $C(0)V$. The amount entering is zero, and the amount leaving each time step is $q(t)C(t)\Delta t$. Thus we should expect that if we add the amount of salt in the tank to the amount that has left the system we should always get an amount that is equal to the original amount. Alternatively, we expect $\int_{t_0}^{t} qC(t)dt + C(t)V - C(0)V = 0$. Adding the following code in the `while(ti <= t_final):` loop:

```
mout += 0.5*(c_old+c_new)*q*dt
mbal  = (c_new*vol+mout-vol*c_init)/(vol*c_init)
```

it is possible to calculate the amount of mass lost (note that we have used the trapezoidal formula to calculate the integral). In the table below the fraction of mass lost relative to the original amount is shown for the various numerical methods.

Δt	h	Euler	RK 2. order	RK 4. order
900	0.9	-0.4500	0.3682	0.0776
500	0.5	-0.2500	0.0833	0.0215
100	0.1	-0.0500	0.0026	0.0008
10	0.01	-0.0050	2.5E-05	8.3E-06

We clearly see from the table that the Runge-Kutta methods performs better than Eulers method, but *all of the methods violates mass balance.*

This might not be a surprise as we know that our numerical solution is always an approximation to the analytical solution. How can we then formulate an algorithm that will respect conservation laws at the discrete level? It turns out that for Eulers method it is not so difficult. Eulers algorithm at the discrete level (see equation (5.6)) is actually a two-step process: first we inject the fresh water while we remove the "old" fluid *and then we mix*. By thinking about the problem this way, it makes more sense to calculate the mass out of the tank as $\sum_k q_k C_k \Delta t_k$. If we in our implementation calculates the mass out of the tank as:

```
mout += c_old*q*dt
mbal  = (c_new*vol+mout-vol*c_init)/(vol*c_init)
```

We easily find that the mass is exactly conserved at every time for Eulers method. The concentration in the tank will of course not be any closer to the analytical solution, but if our mixing tank was part of a much bigger system we could make sure that the mass would always be conserved if we make sure that the mass out of the tank and into the next part of the system was equal to $qC(t)\Delta t$.

5.5 Solving a set of ODE equations

What happens if we have more than one equation that needs to be solved? If we continue with our current example, we might be interested in what would happen if we had multiple tanks in series. This could be a very simple model to describe the cleaning of a salty lake by injecting fresh water into it, but at the same time this lake was connected to two nearby fresh water lakes, as illustrated in figure 5.8. The weakest part of the model is the assumption about complete mixing, in a practical situation we could enforce complete mixing with the salty water in the first tank by injecting fresh water at multiple point in the lake. For the two next lakes, the degree of mixing is not obvious, but salt water is heavier than fresh water and therefore it would sink and mix with the fresh water. Thus if the flow rate was slow, one might imaging that a more or less complete mixing could occur. Our model then could answer questions like, how long time would it take before most of the salt water is removed from the first lake, and how much time would it take before most of the salt water was cleared from the whole system? The answer to these questions would give practical input on how much and how fast one should inject the fresh water to clean up the system. If we had data from an actual

system, we could compare our model predictions with data from the physical system, and investigate if our model description was correct.

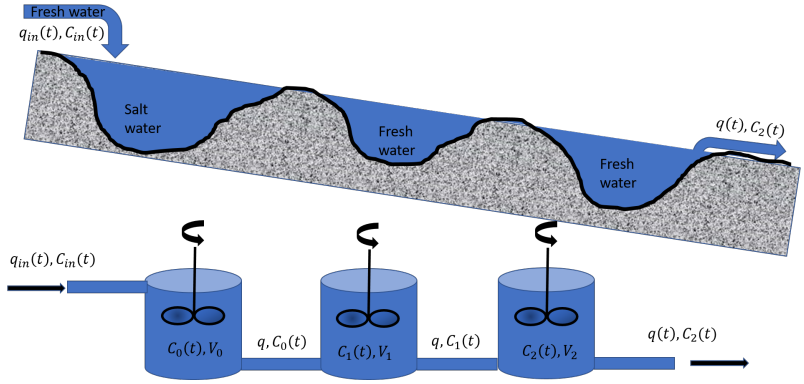

Fig. 5.8 A simple model for cleaning a salty lake that is connected to two lakes down stream.

For simplicity we will assume that all the lakes have the same volume, V. The governing equations follows as before, by assuming mass balance (equation (5.1)):

$$\begin{aligned} C_0(t+\Delta t)\cdot V - C_0(t)\cdot V &= q(t)\cdot C_{\text{in}}(t)\cdot \Delta t - q(t)\cdot C_0(t)\cdot \Delta t, \\ C_1(t+\Delta t)\cdot V - C_1(t)\cdot V &= q(t)\cdot C_0(t)\cdot \Delta t - q(t)\cdot C_1(t)\cdot \Delta t, \\ C_2(t+\Delta t)\cdot V - C_2(t)\cdot V &= q(t)\cdot C_1(t)\cdot \Delta t - q(t)\cdot C_2(t)\cdot \Delta t. \end{aligned} \quad (5.40)$$

Taking the limit $\Delta t \to 0$, we can write equation (5.40) as:

$$V\frac{dC_0(t)}{dt} = q(t)\left[C_{\text{in}}(t) - C_0(t)\right], \quad (5.41)$$

$$V\frac{dC_1(t)}{dt} = q(t)\left[C_0(t) - C_1(t)\right], \quad (5.42)$$

$$V\frac{dC_2(t)}{dt} = q(t)\left[C_1(t) - C_2(t)\right]. \quad (5.43)$$

Let us first derive the analytical solution: Only the first tank is filled with salt water $C_0(0) = C_{0,0}$, $C_1(0) = C_2(0) = 0$, and $C_{\text{in}} = 0$. The solution to equation (5.41) is, as before $C_0(t) = C_{0,0}e^{-t/\tau}$, inserting this equation into equation (5.42) we find:

$$V\frac{dC_1(t)}{dt} = q(t)\left[C_{0,0}e^{-t/\tau} - C_1(t)\right], \tag{5.44}$$

$$\frac{d}{dt}\left[e^{t/\tau}C_1\right] = \frac{C_{0,0}}{\tau}, \tag{5.45}$$

$$C_1(t) = \frac{C_{0,0}t}{\tau}e^{-t/\tau}. \tag{5.46}$$

where we have use the technique of integrating factors[1] when going from equation (5.44) to (5.45). Inserting equation (5.46) into equation (5.43), solving the equation in a similar way as for C_1 we find:

$$V\frac{dC_2(t)}{dt} = q(t)\left[\frac{C_{0,0}t}{\tau}e^{-t/\tau} - C_2(t)\right], \tag{5.47}$$

$$\frac{d}{dt}\left[e^{t/\tau}C_2\right] = \frac{C_{0,0}t}{\tau}, \tag{5.48}$$

$$C_2(t) = \frac{C_{0,0}t^2}{2\tau^2}e^{-t/\tau}. \tag{5.49}$$

The numerical solution follows the exact same pattern as before if we introduce a vector notation. Before doing that, we rescale the time $t \to t/\tau$ and the concentrations, $\hat{C}_i = C_i/C_{0,0}$ for $i = 0, 1, 2$, hence:

$$\frac{d}{dt}\begin{pmatrix}\hat{C}_0(t)\\ \hat{C}_1(t)\\ \hat{C}_2(t)\end{pmatrix} = \begin{pmatrix}\hat{C}_{\text{in}}(t) - \hat{C}_0(t)\\ \hat{C}_0(t) - \hat{C}_1(t)\\ \hat{C}_1(t) - \hat{C}_2(t)\end{pmatrix},$$

$$\frac{d\hat{\mathbf{C}}(t)}{dt} = \mathbf{f}(\hat{\mathbf{C}}, t). \tag{5.50}$$

In figure 5.9 results of an implementation using Runge-Kutta 4. order is shown (see exercises for more details).

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

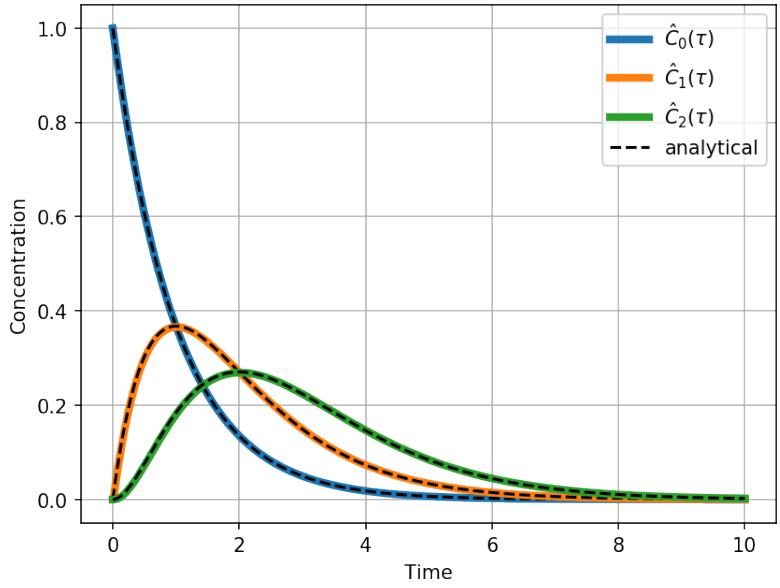

Fig. 5.9 The concentration in the tanks.

5.6 Stiff sets of ODE and implicit methods

As already mentioned a couple of times, our system could be part of a much larger system. To illustrate this, let us now assume that we have two tanks in series. The first tank is similar to our original tank, but the second tank is a sampling tank, 1000 times smaller.

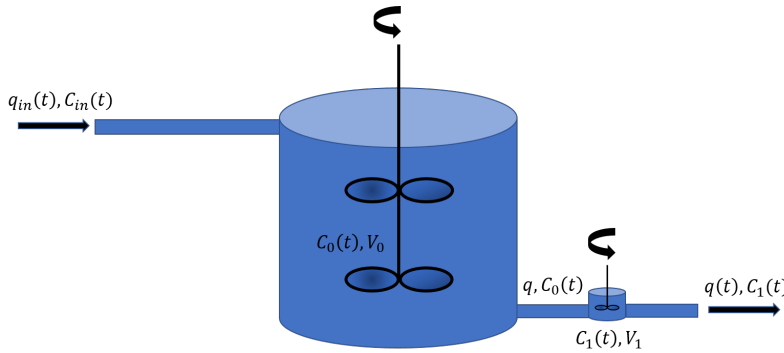

Fig. 5.10 A continuous stirred tank model with a sampling vessel.

The governing equations can be found by requiring mass balance for each of the tanks (see equation (5.1):

$$\begin{aligned} C_0(t+\Delta t)\cdot V_0 - C_0(t)\cdot V_0 &= q(t)\cdot C_{\text{in}}(t)\cdot \Delta t - q(t)\cdot C_0(t)\cdot \Delta t. \\ C_1(t+\Delta t)\cdot V_1 - C_1(t)\cdot V_1 &= q(t)\cdot C_0(t)\cdot \Delta t - q(t)\cdot C_1(t)\cdot \Delta t. \end{aligned} \tag{5.51}$$

Taking the limit $\Delta t \to 0$, we can write equation (5.51) as:

$$V_0 \frac{dC_0(t)}{dt} = q(t)\left[C_{\text{in}}(t) - C_0(t)\right]. \tag{5.52}$$

$$V_1 \frac{dC_1(t)}{dt} = q(t)\left[C_0(t) - C_1(t)\right]. \tag{5.53}$$

Assume that the first tank is filled with seawater, $C_0(0) = C_{0,0}$, and fresh water is flooded into the tank, i.e. $C_{\text{in}} = 0$. Before we start to consider a numerical solution, let us first find the analytical solution: As before the solution for the first tank (equation (5.52)) is:

$$C_0(t) = C_{0,0}e^{-t/\tau_0}, \tag{5.54}$$

where $\tau_0 \equiv V_0/q$. Inserting this equation into equation (5.53), we get:

$$\begin{aligned} \frac{dC_1(t)}{dt} &= \frac{1}{\tau_1}\left[C_{0,0}e^{-t/\tau_0} - C_1(t)\right], \\ \frac{d}{dt}\left[e^{t/\tau_2}C_1\right] &= \frac{C_{0,0}}{\tau_1}e^{-t(1/\tau_0 - 1/\tau_1)}, \qquad (5.55) \\ C_1(t) &= \frac{C_{0,0}}{1-\frac{\tau_1}{\tau_0}}\left[e^{-t/\tau_0} - e^{-t/\tau_1}\right], \qquad (5.56) \end{aligned}$$

where $\tau_1 \equiv V_1/q$.

Next, we will consider the numerical solution. You might think that these equations are more simple to solve numerically than the equations with three tanks in series discussed in the previous section. Actually, this system is much harder to solve with the methods we have discussed so far. The reason is that there are now *two time scales* in the system, τ_1 and τ_2. The smaller tank sets a strong limitation on the step size we can use, because we should never use step sizes larger than a tank volume. Thus if you use the code in the previous section to solve equation (5.52) and (5.53), it will not find the correct solution, unless the step size is lower than 10^{-3}. Equations of this type are known as *stiff*.

Stiff equations

There is no precise definition of "stiff", but it is used to describe a system of differential equations, where the numerical solution becomes unstable unless a very small step size is chosen. Such systems occurs because there are several (length, time) scales in the system, and the numerical solution is constrained by the shortest length scale. You should always be careful on how you scale your variables in order to make the system dimensionless, which is of particular importance when you use adaptive methods.

These types of equations are often encountered in practical applications. If our sampling tank was extremely small, maybe 10^6 smaller than the chemical reactor, then we would need a step size of the order of 10^{-8} or lower to solve the system. This step size is so low that we easily run into trouble with round off errors in the computer. In addition the simulation time is extremely long. How do we deal with this problem? The solution is actually quite simple. The reason we run into trouble is that we require that the concentration leaving the tank must be a small perturbation of the old one. This is not necessary, and it is best illustrated with Eulers method. As explained earlier Eulers method can be viewed as a two step process: first we inject a volume (and remove an equal amount: $qC(t)\Delta t$), and then we mix. Clearly when we try to remove more than what is left, we run into trouble. What we want to do is to remove or flood much more than one tank volume through the tank during one time step, this can be achieved by $q(t)C(t)\Delta t \rightarrow q(t+\Delta t)C(t+\Delta t)\Delta t$. The term $q(t+\Delta t)C(t+\Delta t)\Delta t$ now represents *the mass out of the system during the time step* Δt.

The methods we have considered so far are known as *explicit*, whenever we replace the solution in the right hand side of our algorithm with $y(t+\Delta t)$ or (y_{n+1}), the method is known as *implicit*. Implicit methods are always stable, meaning that we can take as large a time step that we would like, without getting oscillating solution. It does not mean that we will get a more accurate solution, actually explicit methods are usually more accurate.

Explicit and Implicit methods

Explicit methods are often called *forward* methods, as they use only information from the previous step to estimate the next value. The explicit methods are easy to implement, but get into trouble if the step size is too large. Implicit methods are often called *backward* methods as the next step cannot be calculated directly from the previous solution, usually a non-linear equation has to be solved. Implicit methods are generally much more stable, but the price is often lower accuracy. Many commercial simulators uses implicit methods extensively because they are stable, and stability is often viewed as a much more important criterion than numerical accuracy.

Let us consider our example further, and for simplicity use the implicit Eulers method:

$$\begin{aligned} C_{0n+1}V_0 - C_{0n}V_0 &= q(t+\Delta t)C_{\text{in}n+1}\Delta t - q(t+\Delta t)C_{0n+1}\Delta t. \\ C_{1n+1}V_1 - C_{1n}V_1 &= q(t+\Delta t)C_{0n+1}\Delta t - q(t+\Delta t)C_{1n+1}\Delta t. \end{aligned} \quad (5.57)$$

This equation is equal to equation (5.51), but the concentrations on the right hand side are now evaluated at the next time step. The immediate problem is now that we have to find an expression for C_{n+1} that is given in terms of known variables. In most cases one needs to use a root finding method, like Newtons method, in order to solve equation (5.57). In this case it is straight forward to show:

$$\begin{aligned} C_{0n+1} &= \frac{C_{0n} + \frac{\Delta t}{\tau_0}C_{\text{in}n+1}}{1+\frac{\Delta t}{\tau_0}}, \\ C_{2n+1} &= \frac{C_{1n} + \frac{\Delta t}{\tau_1}C_{0n+1}}{1+\frac{\Delta t}{\tau_1}}. \end{aligned} \quad (5.58)$$

In figure 5.11 the result of the implementation is shown, note that quite large step sizes can be used without inducing non physical results.

Exercise 5.1: Truncation Error in Eulers Method

In the following we will take a closer look at the adaptive Eulers algorithm and show that the constant c is indeed the same in equation (5.19) and (5.20). The true solution $y(t)$, obeys the following equation:

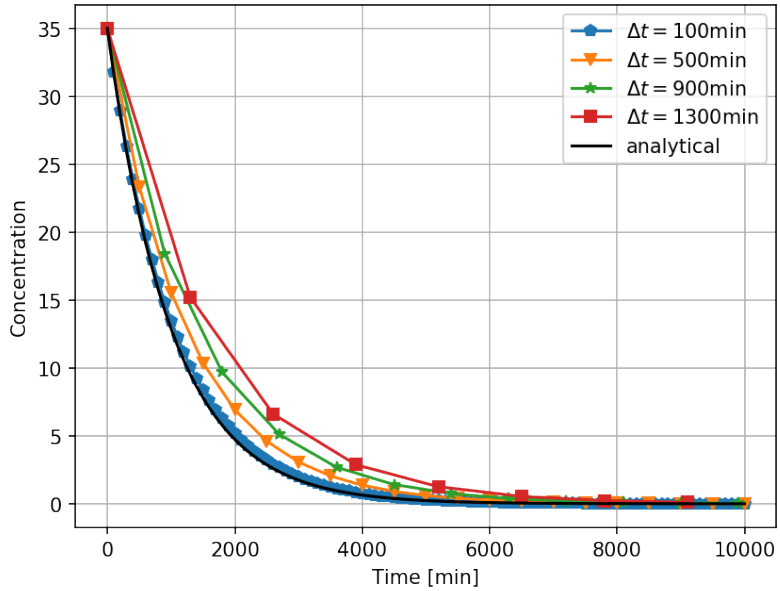

Fig. 5.11 The concentration in the tanks for $h = 0.01$.

$$\frac{dy}{dt} = f(y,t), \tag{5.59}$$

and Eulers method to get from y_0 to y_1 by taking one (large) step, h is:

$$y_1^* = y_0 + hf(y_0, t_0), \tag{5.60}$$

We will also assume (for simplicity) that in our starting point $t = t_0$, the numerical solution, y_0, is equal to the true solution, $y(t_0)$, hence $y(t_0) = y_0$.

a) Show that when we take one step of size h from t_0 to $t_1 = t_0 + h$, $c = y''(t_0)/2$ in equation (5.19).

Answer. The local error, is the difference between the numerical solution and the true solution:

$$\begin{aligned}\epsilon^* = y(t_0+h) - y_1^* &= y(t_0) + y'(t_0)h + \frac{1}{2}y''(t_0)h^2 + \mathcal{O}(h^3) \\ &- [y_0 + hf(y_0, t_0 + h)]\,,\end{aligned} \tag{5.61}$$

where we have used Taylor expansion to expand the true solution around t_0, and equation (5.60). Using equation (5.59) to replace $y'(t_0)$ with $f(y_0, t_0)$, we find:

$$\epsilon^* = y(t_0 + h) - y_1^* = \frac{1}{2}y''(t_0)h^2 \equiv ch^2, \tag{5.62}$$

hence $c = y''(t_0)/2$.

b) Show that when we take two steps of size $h/2$ from t_0 to $t_1 = t_0 + h$, Eulers algorithm is:

$$y_1 = y_0 + \frac{h}{2}f(y_0, t_0) + \frac{h}{2}f(y_0 + \frac{h}{2}f(y_0, t_0), t_0 + h/2). \tag{5.63}$$

Answer.

$$y_{1/2} = y_0 + \frac{h}{2}f(y_0, t_0), \tag{5.64}$$

$$y_1 = y_{1/2} + \frac{h}{2}f(y_{1/2}, t_0 + h/2), \tag{5.65}$$

$$y_1 = y_0 + \frac{h}{2}f(y_0, t_0) + \frac{h}{2}f(y_0 + \frac{h}{2}f(y_0, t_0), t_0 + h/2). \tag{5.66}$$

Note that we have inserted equation (5.64) into equation (5.65) to arrive at equation (5.66).

c) Find an expression for the local error when using two steps of size $h/2$, and show that the local error is: $\frac{1}{2}ch^2$

Answer.

$$\epsilon = y(t_0 + h) - y_1 = y(t_0) + y'(t_0)h + \frac{1}{2}y''(t_0)h^2 + \mathcal{O}(h^3) - \left[y_0 + \frac{h}{2}f(y_0, t_0) + \frac{h}{2}f(y_0 + \frac{h}{2}f(y_0, t_0), t_0 + h/2)\right]. \tag{5.67}$$

This equation is slightly more complicated, due to the term involving f inside the last parenthesis, we can use Taylor expansion to expand it about (y_0, t_0):

$$f(y_0 + \frac{h}{2}f(y_0, t_0), t_0 + h/2) = f(y_0, t_0) + \frac{h}{2}\left[f(y_0, t_0)\left.\frac{\partial f}{\partial y}\right|_{y=y_0, t=t_0} + \frac{h}{2}\left.\frac{\partial f}{\partial t}\right|_{y=y_0, t=t_0}\right] + \mathcal{O}(h^2). \tag{5.68}$$

It turns out that this equation is related to $y''(t_0, y_0)$, which can be seen by differentiating equation (5.59):

$$\frac{d^2y}{dt^2} = \frac{df(y,t)}{dt} = \frac{\partial f(y,t)}{\partial y}\frac{dy}{dt} + \frac{\partial f(y,t)}{\partial t} = \frac{\partial f(y,t)}{\partial y} f(y,t) + \frac{\partial f(y,t)}{\partial t}. \tag{5.69}$$

Hence, equation (5.68) can be written:

$$f(y_0 + \frac{h}{2} f(y_0, t_0), t_0 + h/2) = f(y_0, t_0) + \frac{h}{2} y''(t_0, y_0), \tag{5.70}$$

hence the truncation error in equation (5.67) can finally be written:

$$\epsilon = y(t_1) - y_1 = \frac{h^2}{4} y''(y_0, t_0) = \frac{1}{2} c h^2, \tag{5.71}$$

Solution. The local error, is the difference between the numerical solution and the true solution:

$$\begin{aligned} \epsilon^* = y(t_0 + h) - y_1^* = y(t_0) + y'(t_0)h + \frac{1}{2} y''(t_0)h^2 + \mathcal{O}(h^3) \\ - \left[y_0 + h f(y_0, t_0 + h) \right], \end{aligned} \tag{5.72}$$

where we have used Taylor expansion to expand the true solution around t_0, and equation (5.60). Using equation (5.59) to replace $y'(t_0)$ with $f(y_0, t_0)$, we find:

$$\epsilon^* = y(t_0 + h) - y_1^* = \frac{1}{2} y''(t_0)h^2 \equiv c h^2, \tag{5.73}$$

where we have ignored terms of higher order than h^2, and defined c as $c = y''(t_0)/2$. Next we take two steps of size $h/2$ to reach y_1:

$$y_{1/2} = y_0 + \frac{h}{2} f(y_0, t_0), \tag{5.74}$$

$$y_1 = y_{1/2} + \frac{h}{2} f(y_{1/2}, t_0 + h/2), \tag{5.75}$$

$$y_1 = y_0 + \frac{h}{2} f(y_0, t_0) + \frac{h}{2} f(y_0 + \frac{h}{2} f(y_0, t_0), t_0 + h/2). \tag{5.76}$$

Note that we have inserted equation (5.74) into equation (5.75) to arrive at equation (5.76). The truncation error in this case is, as before:

$$\begin{aligned} \epsilon = y(t_0 + h) - y_1 = y(t_0) + y'(t_0)h + \frac{1}{2} y''(t_0)h^2 + \mathcal{O}(h^3) \\ - \left[y_0 + \frac{h}{2} f(y_0, t_0) + \frac{h}{2} f(y_0 + \frac{h}{2} f(y_0, t_0), t_0 + h/2) \right]. \end{aligned} \tag{5.77}$$

This equation is slightly more complicated, due to the term involving f inside the last parenthesis, we can use Taylor expansion to expand it about (y_0, t_0):

$$f(y_0 + \frac{h}{2} f(y_0, t_0), t_0 + h/2) = f(y_0, t_0) + \frac{h}{2}\left[f(y_0, t_0) \left.\frac{\partial f}{\partial y}\right|_{y=y_0, t=t_0} + \left.\frac{\partial f}{\partial t}\right|_{y=y_0, t=t_0} \right] + \mathcal{O}(h^2). \tag{5.78}$$

It turns out that this equation is related to $y''(t_0, y_0)$, which can be seen by differentiating equation (5.59):

$$\frac{d^2 y}{dt^2} = \frac{df(y,t)}{dt} = \frac{\partial f(y,t)}{\partial y}\frac{dy}{dt} + \frac{\partial f(y,t)}{\partial t} = \frac{\partial f(y,t)}{\partial y} f(y,t) + \frac{\partial f(y,t)}{\partial t}. \tag{5.79}$$

Hence, equation (5.78) can be written:

$$f(y_0 + \frac{h}{2} f(y_0, t_0), t_0 + h/2) = f(y_0, t_0) + \frac{h}{2} y''(t_0, y_0), \tag{5.80}$$

hence the truncation error in equation (5.77) can finally be written:

$$\epsilon = y(t_1) - y_1 = \frac{h^2}{4} y''(y_0, t_0) = \frac{1}{2} c h^2, \tag{5.81}$$

Monte Carlo Methods

6

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.