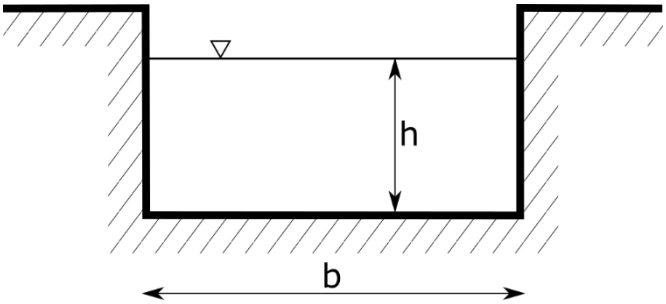
3. NUMERICAL ERRORS

INTORDUCTION TO NUMERICAL METHODS

Certain tasks and problems cannot be solved using traditional analytical mathematical methods, or can only be solved with great difficulty. In these cases, numerical methods can be used. Analytical solutions are exact solutions, where the variables included in the task can be expressed with closed formulas. Such is the formula for solving a quadratic equation. The numerical solution is only an approximate numerical value of the real solution. Although the numerical solution is only an approximation, its value can be very accurate. Most numerical methods are local, iterative procedures where the solution is gradually approximated until the desired accuracy is reached. Let's look at a civil engineering example, which cannot be solved easily analytically!

The design of a channel is a common task in hydraulics. Depending on the shape, material, slope, width and water level of the open-surface channel, the amount of transported water can be deduced. For example, let's look at the formula for the water flow rate of a channel with a free surface and a rectangular cross section!

$$Q = \frac{\sqrt{s}}{n} \cdot \frac{(b \cdot h)^{\frac{5}{3}}}{(b + 2 \cdot h)^{\frac{2}{3}}}$$



where Q – water flow rate, n – Manning's coefficient, s - slope, b – channel width, h – water flow depth.

This is an exact formula for Q, but if we are interested in the water depth at which the standard Q at that area can flow through, then we can no longer express it explicitly. Previously, different tables or graphs were used for this task. Although we cannot produce the solution analytically, since the spread of modern computers, we can give an approximate value of the solution with a predetermined accuracy using numerical methods. This means that if we substitute back the solution for the height to the equation for Q, we will not get back the exact value of the water flow rate (Q), but we will be very close to it.

Numerical techniques for solving such problems have been developed for centuries, but their application was very complicated before the spread of today's computers. Calculations made by hand or with a mechanical calculator were very time-consuming and easy to miscalculate. These techniques could only spread with the emergence of computer technology, since it is no longer a problem to perform many repetitive, complicated calculations in a short time.

When solving an engineering problem, the task, variables, and conditions (boundary values, initial values) must first be defined. After that, the physical model must be set up for the problem, it can be the water flow formula, mass attraction, Newton's laws, etc. It must be decided whether the task can be solved analytically or only numerically. It must be investigated whether acceptable simplifications (e.g. linearization) can be made for the analytical solution. Sometimes, even in the case of numerical

calculations, simplifications must be used if the model is too complicated to solve the task within a foreseeable time (see, for example, weather forecasts).

Even when using numerical calculations, it is necessary to choose which method to use. We can choose from a wide variety of developed methods for each type of task. The methods may differ in accuracy, computational requirements, and programming complexity. The selected algorithm must also be implemented in the programming language we use. When using Matlab/Octave, in most cases we do not have to program the algorithms, since there are many built-in numerical methods in these softwares, but it is also advisable to get to know their background so that we can apply them correctly.

After solving the task, we have to check the solution in some way. In the case of a non-linear equation, this can be done, for example, by back substitution. In more complicated cases, for example when solving differential equations, the numerical solution can be compared with a known solution of a similar problem, or it can be solved using different methods and their differences can be examined.[1]

ROUND-OFF ERROR, FLOATING POINT NUMBER REPRESENTATION

Since we will solve our tasks with the help of a computer, we need to know the limitations of computers, we need to know how numbers are stored, what errors can arise from the storage method or the choice of algorithms. Let's look at the following simple examples!

Let's try the following in Matlab:

```
> x1 = 0.3, x2 = 0.1+0.1+0.1
```

Are these two numbers equal? Obviously, yes. Let's check it in Matlab!

```
> x1==x2
```

In response, we received 0 (false)! What could be the reason? Why can't Matlab solve such an obvious, simple example? To understand, let's examine how the computer stores the numbers!

- The default and most common form of storing numbers is double precision floating point.
- This is IEEE standardized format (IEEE 754).
- All numbers are stored in binary format using 64 bits (0-63).

Number representation:

$$(-1)^s \cdot m \cdot 2^{(e-1023)},$$

[1] See: Amos Gilat, Vish Subramaniam (2011): Numerical Methods, An Introduction with Applications Using MATLAB (SI Version), John Wiley & Sons (Asia)

where *s* is the sign bit (1 bit), *m* is the mantissa (significant digits stored in 52 bits) and *e* is the exponent (11 bits) (s+m+e=>1+11+52 = 64 bits).

It is not possible to accurately represent all the real numbers (because there are infinite). Let's see the number e.g. 0.1 in the binary number system! In this system, 0.1 is an infinite periodic fractional number, of which only the first 52 bits are used, the rest are discarded. This is the round-off error.

$$\frac{1}{10} = \frac{1}{2^4} + \frac{1}{2^5} + \frac{1}{2^8} + \frac{1}{2^9} + \frac{1}{2^{12}} + \frac{1}{2^{13}} + \frac{1}{2^{16}} + \cdots$$

0.1 in binary → 0.0001100110011… (infinite periodic fraction)

Added up this infinite periodic fraction three times will result in a round-off error, so the two values will not be exactly the same within the precision used in the number representation.

Let's see what the value of $tan\left(\frac{\pi}{2}\right)$ is in Matlab? The tangent function is not interpreted at $\frac{\pi}{2}$, it goes to infinity. What happens if we want to calculate this in Matlab? Are we getting an error message?

```
> tan(pi/2)
```

What was the result in Matlab? $1.6331 \cdot 10^{16}$, which is not infinite, just a very large number. Why? Because due to the round-off error, we cannot represent the value of $\frac{\pi}{2}$ exactly.

What exactly is round-off error (δ)?

$$1 = 1 + \delta, if\ \delta < \varepsilon_m,$$

where δ is the round-off error and ε is the machine precision or machine epsilon.

Machine precision is the smallest representable distance between 1 and the next smallest number. In Matlab, it can be queried with the **eps** variable or the **eps()** function.

```
> eps
```

By default, its value is $\approx 2 \cdot 10^{-16}$. Let's add 10^{-17} to 1 (which is less than machine epsilon)!

```
> a = 1
> b = 1 + 1·10^-17 % (less than machine epsilon))
> a == b  % -> 1 (true)
```

Now add 10^{-14} (which is greater than the machine epsilon) to two different numbers, the first should be 1, the second 12345.

```
> a = 1; b = 12345; c = 1e-14;
> a == a+c % 0 (false)
```

```
> b == b+c % 1 (true)
```

Why is it that in one case a value greater than the machine epsilon could be added to the number, but not in the other case? The answer to this is that the default value of machine precision is the distance from 1, but its value varies with the magnitude of the number. Let's look at it!

```
> eps(a) % 2.2204e-16
> eps(b) % 1.8190e-12
> eps(1e20) % 16384
```

This means that the smallest number that can be stored after 10^{20} is 16384 larger. If we add a number 16384/2=8192 or less to 10^{20}, the value of the number does not change, if it is greater, it does (due to rounding, if we add a number greater than 16384/2, it is already rounded up).

```
> 1e20 == 1e20+8192 % true
> 1e20 == 1e20+8193 % false
```

CANCELLATION ERROR, OVERFLOW ERROR

Let's look at another typical rounding error, the cancellation error, when numbers of nearly the same magnitude are subtracted from each other and this may cause extreme loss of accuracy. Let's look at the following example in Matlab:

```
> x1 = 10.000000000000004 % 14 zeros after the decimal point before the
  digit 4
> y1 = 10.00000000000004 % % 13 zeros after the decimal point before
  the digit 4
> % or: x1 = 4e-15 + 10; y1 = 4e-14 + 10;
> (y1-10)/(x1-10) % expected result: 10
```

The expected result: 0.000000000000004 / 0.00000000000004 = 10, but the result is: 11.5! The cancellation error caused a very large loss of accuracy in this case.

We have seen the round-off errors resulting from machine precision, but it is also important to know the range of numbers that can be represented, because exceeding this can also lead to serious errors! This is called an overflow error.

```
> realmin % 2.2251e-308
> realmax % 1.7977e+308
```

Why is it important to know about numerical errors? It is worth looking at some cases among the disasters caused by numerical errors! During the Gulf War in 1991, a Patriot anti-aircraft missile missed an Iraqi Scud missile, killing 28 people. The reason was a numerical error. The system multiplied the times measured in tenths of a second by 1/10 to get seconds. Since 1/10 could not be accurately represented in binary format, a small round-off error occurred and the error accumulated after performing the operation several times. In the case of 100 hours of operation, the difference was 0.34 seconds, during which a Scud missile travels more than half a kilometer (at a speed of 1,676 m/s).

A numerical error (an overflow error) caused the explosion of ESA's Ariane 5 rocket 40 seconds after its launch in 1996! („Famous number computing errors” - https://blog.penjee.com/famous-number-computing-errors/, „Disasters due to rounding error” - https://web.ma.utexas.edu/users/arbogast/misc/disasters.html). It might also

be interesting to look at the following website, which contains a larger collection of problems caused by software bugs (many of which are numerical errors): https://www5.in.tum.de/persons/huckle/bugse.html

TRUNCATION ERROR

We have already seen the effect of the round-off error resulting from the finite representation of numbers, and also that it is important to avoid subtracting numbers of nearly the same size if possible.

Another important error occurs when, instead of the exact mathematical expression, its approximation is used during numerical calculations, for example: Taylor series approximation, using a difference quotient instead of a derivative, etc. The error caused by the approximation is the truncation error. The problem usually arises when a complicated problem that cannot be solved symbolically is replaced by a simpler problem that can be handled more easily with a computer.

A good example of this is the Taylor series approximation, where the more terms we take into account, the smaller the truncation error will be. Now let's look at the approximation of e^x with a Taylor series with 4 terms: $e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!}$

```
> x=1
> f = exp(x) % 2.7183 - real value
> g = 1 + x + x^2/2 + x^3/6 % 2.6667 - value with a truncation error
```

Rounding and truncation errors together give the total error!

round-off error + truncation error = total error

ABSOLUTE AND RELATIVE ERROR

Errors can be grouped in several ways, the actual deviation from the exact value is called an absolute error, but in reality it is often not the best way to represent the magnitude of the error, it is advisable to introduce the concept of relative error as well. Let the approximation of a real number x be $\tilde{x}$.

We call the actual deviation the absolute error: $\Delta = |x - \tilde{x}|$

The relative error is the absolute error divided by the value of x: $\varepsilon = \frac{|x - \tilde{x}|}{|x|}$

When x is a value around one, there is no significant difference between the two errors, however, when x>>1, the relative error better reflects the significance of the error. Let's look at an example of this! Let there be two distances (t_1, t_2) that we estimate $(\tilde{t_1}, \tilde{t_1})$.

$t_1 = 1000\ m; \quad \tilde{t_1} = 900\ m;$ $\Delta = 100\ m; \quad \varepsilon = 10\ \%;$	$t_2 = 200\ m; \quad \tilde{t_1} = 100\ m;$ $\Delta = 100\ m; \quad \varepsilon = 50\ \%;$

The absolute error is 100 m in both cases, but it is obvious that the first estimate is much more accurate than the second and this is also reflected in the size of the relative errors (10% and 50%).

STABILITY AND CONDITION NUMBER

Having seen that we have to live with numerical errors in calculations, the next important question is how do we deal with them? Can we trust the results? For this, we need to get to know two more concepts, the sensitivity/condition number of a specific problem and the stability of the algorithm.

- A mathematical problem is well-conditioned if the result changes slightly due to small change in the input parameters.
- An algorithm is numerically stable if the result changes slightly due to small change in the input parameters.

The accuracy depends on the conditionality of the problem and the stability of the algorithm. Applying a stable algorithm to an ill-conditioned problem or applying an unstable algorithm to a well-conditioned problem can cause inaccuracy.

CONDITION NUMBER

Let's solve the following system of linear equations!

$$6x_1 - 2x_2 = 10$$

$$11.5x_1 - 3.85x_2 = 17$$

The system of equations is in matrix form ($A \cdot x = b$):

$$A = \begin{pmatrix} 6 & -2 \\ 11.5 & -3.85 \end{pmatrix}; \quad b = \begin{pmatrix} 10 \\ 17 \end{pmatrix}$$

Let's solve this problem in Matlab. The solution is: $x = A^{-1} \cdot b$. In Matlab, the **inv** command calculates the inverse of a matrix. Let's solve the task!

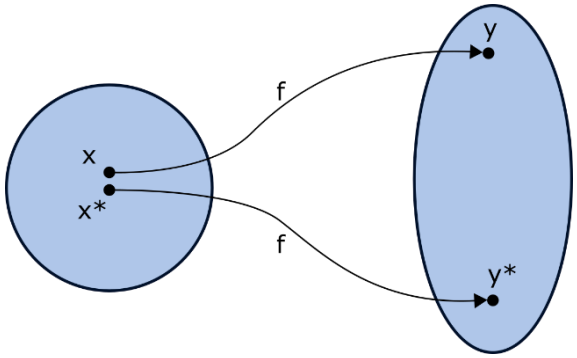
```
> A = [6,-2; 11.5,-3.85]; b = [10; 17];
> x = inv(A)*b % 45.0000; 130.0000
```

The solution was 45 and 130. Let's slightly change the coefficient of x_2 in the second equation from -3.85 to -3.84 and solve the problem again!

```
> A = [6,-2; 11.5,-3.84]
> x = inv(A)*b % 110.0000; 325.0000
```

Now the solution is 110 and 325. We only changed the system of equations a little, but the change in the final result was huge! We would have expected that since the two inputs are very close to each other, the solutions would be similar. It didn't happen that way. What could have caused this?

There are matrices that are very sensitive to small changes in the input. This sensitivity can be measured by the condition number of the matrix. The condition number (κ) establishes a relationship between the relative error of the output and the relative error of the input. The larger this number is, the greater the change in the output will be for a small change in the input.



$$\kappa = \left| \frac{\frac{f(x)-f(\tilde{x})}{f(\tilde{x})}}{\frac{x-\tilde{x}}{\tilde{x}}} \right| = \left| \frac{\tilde{x}}{f(\tilde{x})} \cdot \frac{f(x)-f(\tilde{x})}{x-\tilde{x}} \right| = \left| \frac{\tilde{x} \cdot f'(\tilde{x})}{f(\tilde{x})} \right|$$

Let's look at the condition number of this matrix!

```
> cond(A) % 4.6749e+03
```

The result was 4674.9. The larger this number, the more uncertain the solution, because a small error in the input is magnified by the same amount in the output This is very important in engineering, where input measurements and constants are mostly only approximations and may contain errors.

EXAMPLE OF AN UNSTABLE ALGORITHM

Let's look at the solution of the following quadratic equation!

$$x^2 - 100.0001\,x + 0.01 = 0$$

The exact solution is: $x_1 = 100; x_2 = 0.0001$ (with Vieta's formulas[2]). A solution can also be given with the well-known quadratic formula:

$$x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a}; \qquad x_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a};$$

Let's solve it in Matlab! For more accurate results, change the display to more decimal places!

```
> format long;
> a = 1; b = -100.0001; c = 0.01;
> D = sqrt(b^2 - 4*a*c) % 99.999899999999997
> x1 = (-b + D)/(2*a) % 100
> x2 = (-b - D)/(2*a) % 1.000000000033197e-04
```

We did not get an exact result for x_2 due to the rounding error. Since *b* is negative, so in this case two very close values had to be subtracted from each other in the numerator, the canceling error occurred here as well!

In many cases, when the mathematical expression contains the difference between two close expressions, the problem can be transformed into a form that is less sensitive to rounding errors. In case of the formula for x_2, we can do this by multiplying the equation by $(-b + \sqrt{b^2 - 4ac})/(-b + \sqrt{b^2 - 4ac})$:

$$x_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a} \cdot \frac{-b + \sqrt{b^2 - 4ac}}{-b + \sqrt{b^2 - 4ac}} = \frac{2c}{-b + \sqrt{b^2 - 4ac}};$$

Let us now use the latter expression for the solution:

```
> x2m = (2*c)/(-b+D) % 1.000000000000000e-04
```

Now we got the expected result.

[2] Vieta's formulas for $a \cdot x^2 + b \cdot x + c = 0$ are: $x_1 + x_2 = -b/a;\ x_1 \cdot x_2 = c/a$

EXAMPLE OF A STABLE ALGORITHM

Let's look at another example of the importance of choosing an algorithm! We can approximate the value of e^{-x} in two ways:

$$e^{-x} = 1 - x + \frac{x^2}{2!} - \frac{x^3}{3!} + \cdots = f(x)$$

$$e^{-x} = \frac{1}{e^x} = \frac{1}{1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots} = g(x)$$

Let's calculate the value of the function at x = 8.3 using the two methods given above! First check the emx.m function, which gives two outputs for the two approximations for n members, at x locations!

```
function [f g] = emx(x,n)
    f = 1; % first approximation: 1 - x + x^2/2 - x^3/6 + ...
    p = 1; % first approximation to the denominator (1 + x + x^2/2 +
x^3/6 + ...)
    for i=1:n
        s = x^i/factorial(i);
        f = f +(-1)^i*s;
        p = p + s;
        g = 1 / p;
    end
end
```

The exact value of the solution is: exp(-8.3) = 2.4852e-04

Let's try the previous function in the case of n = 10, 20, 30! Let's use a format with fewer digits, that will be enough for now.

```
format short
megoldas = exp(-8.3) % Exact value: 2.4852e-04
[f g] = emx(8.3,10) % approx. for 10 members: f=188.0344,
g=3.1657e-04
[f g] = emx(8.3,20) % approx. for 20 members: f=0.2833,
g=2.4856e-04
[f g] = emx(8.3,30) % approx. for 30 members: f=2.5151e-04,
g=2.4852e-04
```

The results:

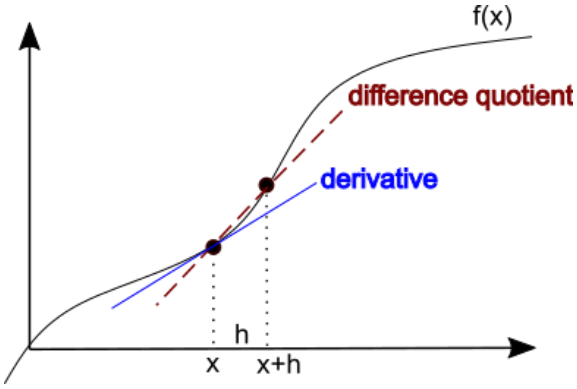
exact: 2.4852e-04	n=10	n=20	n=30
f (approximation)	188.0344	0.2833	2.5151e-04
g (approximation)	3.1657e-04	2.4856e-04	2.4852e-04

We see that the second of the two algorithms approaches the exact value much faster, so it really does matter which method we use to solve the problem!

TOTAL ERROR

Let's look at an example where both truncation and rounding errors occur. Not only the Taylor series approximation contains a truncation error, but also the numerical integration or the approximation of the derivative with the difference quotient. Let's look at an example of the latter:

$$f'(x) \approx \frac{f(x+h)-f(x)}{h}$$



The upper limit of the truncation error can be estimated. Determine the upper limit of the truncation error in the case of approximating the derivative of the following function[3] with the difference quotient, at *x*=2!

$$y = x^3$$

Let's approximate the derivative numerically with the difference quotient!

$$f'(x) \approx \frac{f(x+h)-f(x)}{h} = \frac{(2+h)^3-2^3}{h}$$

The truncation error can be estimated based on the Taylor series approximation:

$$f(x+h) = f(x) + f'(x)\cdot h + \frac{f''(\xi)}{2}\cdot h^2$$

, where $x < \xi < x+h$. Therefore, rearranging the equation, the following is true:

$$\left|\frac{f(x+h)-f(x)}{h} - f'(x)\right| \le \frac{f''(\xi)}{2}\cdot h$$

On the left-hand side of the equation above, we find the truncation error (the difference between the approximation and the actual value), and on the right-hand side there is an upper limit for this, at which the error will certainly be smaller. Let us denote this error by *R*, and the upper limit by ε. In our case, we know exactly the first and the second derivative ($y' = 3\,x^2 = 12$ and $y'' = 6x$), so the following is true for the truncation error

$$R(h) = \left|\frac{(2+h)^3-2^3}{h} - 12\right| \le \frac{f''(\xi)}{2}\cdot h = \frac{6\,\xi\,h}{2} = 3\,\xi\,h$$

The largest error in the estimation occurs when $\xi = x+h$, in the case of $x = 2$, the estimated upper limit of the truncation error as a function of the step interval *h* will be as follows:

$$\varepsilon(h) = 3\,(2+h)\,h$$

From the formula above, it can be seen, as expected, that the smaller the step *h* is used to divide the function for the calculation of the difference quotient, the smaller the truncation error will be.

[3] Based on Béla Paláncz's collection of examples of numerical methods

Let's plot the estimated upper limit of the approximation of the previous derivative and calculate the actual error as a function of the change in the step interval! Let's use the following step intervals:

$$hi = 1, 10^{-1}, 10^{-2}, \cdots, 10^{-15}$$

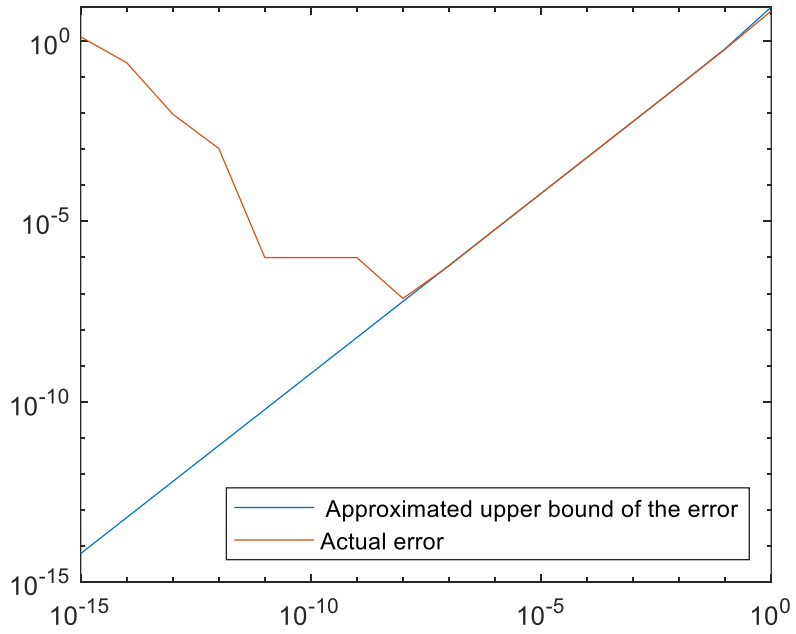
```
> format long
> n = 0:-1:-15, hi = 10.^n
```

Enter both the estimated upper limit and the actual error as a one-line, anonymous function! (Pay attention to the use of .*, ./, .^ operators, as element-by-element operations on vectors are now required.)

```
> e = @(h) 3*(2 + h).*h; % estimated upper limit
> R = @(h) abs(((2 + h).^3 - 8)./h - 12) % actual error
```

Let's calculate the value of the two functions for the different step intervals and plot the results in the log-log coordinate system (this can be done with the **loglog** command)!

```
> figure(1)
> loglog(hi,e(hi)); hold on
> loglog(hi,R(hi))
> legend('Approximated upper bound of the error','Actual error',...
>     'Location','SE')
```



What do we see in the figure above? As the step interval decreases, so does the estimated upper limit of the truncation error and for a while the actual error accordingly. But after a point (somewhere around 10^{-8}) the actual error suddenly starts to increase again. What could be the reason for this? This results from the fact that the total error is the sum of the truncation and rounding errors. The truncation error dominates up to a value of roughly 10^{-8}, after which the rounding error starts to increase strongly. This phenomenon plays a very important role, for example, in the numerical solution of differential equations!

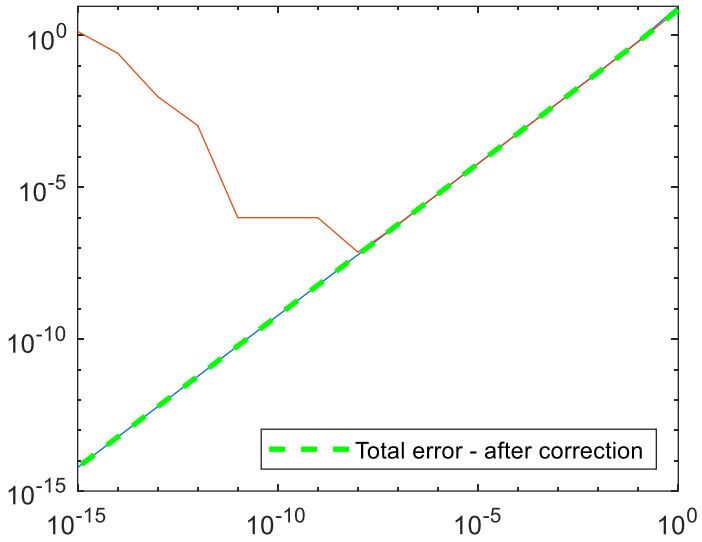
Of course, similarly to what we saw before, here you can also choose an algorithm that is less sensitive to rounding errors, since the main error here is also the cancellation error. In the numerator of the difference quotient, two nearly identical numbers are subtracted from each other, the smaller the step interval *h*, the smaller the difference between the two numbers. Now we can easily simplify the difference quotient:

$$\frac{(2+h)^3 - 2^3}{h} = 12 + 6\,h + h^2$$

Based on this, the corrected error is:

$$R(h) = |12 + 6\,h + h^2 - 12| = |6\,h + h^2|$$

```
> R2 = @(h) abs(6*h+h.^2);
> fgr1 = loglog(h,R2(h),'g--','LineWidth',2)
> legend(fgr1,'Total error - after correction')
```



Here we simplified the difference ratio manually, of course such simplifications are also possible in Matlab, but they already belong to the topic of symbolic calculations, which is part of the Symbolic Math Toolbox.

During symbolic calculations, we do not calculate with numbers, but with variables. To do this, you must first define the symbolic variables in Matlab, with the **syms** command. If we call a function with symbolic variables, we get a symbolic expression that can be simplified with the **simplify** command. The result of the **simplify** command will be a symbolic expression into which numbers cannot be substituted, only if we convert the expression back into a Matlab function with the **matlabFunction** command.

```
> syms h
> Rsym = simplify(R(h)) % Rsym = abs(h*(h + 6))
> R2 = matlabFunction(Rsym) % R2 = @(h)abs(h.*(h+6.0))
```

We can also see in Workspace that the value of Rsym is symbolic. Let's see what happens if we try to substitute a specific value into a function or a symbolic expression!

```
> R2(1e-1) % 0.610000000000000
> Rsym(1e-1)
Subscript indices must either be real positive integers or logicals.
```

In the second case, we receive an error message, since a value can only be substituted into a symbolic expression using the **subs** command.

```
> subs(Rsym,1e-5) % 600001/10000000000
```

NEW FUNCTIONS USED IN THE CHAPTER

eps	-	Magnitude of machine epsilon/machine accuracy
realmin	-	The smallest number that can be represented (in the case of double type)
realmax	-	The largest number that can be represented (in the case of double type)
factorial	-	Factorial, n!
inv	-	Matrix inverse
cond	-	Condition number
loglog	-	Representation on a logarithmic scale (on both axes)
syms	-	Defining symbolic variables and expressions
simplify	-	Simplifying symbolic expressions
matlabFunction	-	Converting symbolic expressions into functions