1 Numerical error

1.1 Types of error

The term "error" is going to appear throughout this book in different contexts. The varieties of error we will be concerned with are:

- Experimental error. We may wish to calculate some function $y(x_1, \ ..., \ x_n)$, where the quantities x_i are measured. Any such measurement has associated errors, and they will affect the accuracy of the calculated y.
- Roundoff error. Even if x were measured exactly, odds are it cannot be represented exactly in a digital computer. Consider π, which cannot be represented exactly in decimal form. We can write $\pi \approx 3.1416$, by rounding the exact number to fit 5 decimal figures. Some roundoff error occurs in almost every calculation with real numbers, and controlling how strongly it impacts the final result of a calculation is always an important numerical consideration.
- Approximation error. Sometimes we want one thing but calculate another, intentionally, because the other is easier or has more favorable properties. For example, one might choose to represent a complicated function by its Taylor series. When substituting expressions that are not mathematically identical we introduce approximation error.

Experimental error is largely outside the scope of numerical treatment, and we'll assume here, with few exceptions, that it's just something we have to live with. Experimental error plays an important role in data fitting, which will be described at length in Chapter 8. Sampling error in statistical processes can be thought of as a type of experimental error, and this will be discussed in Chapter 11.

Controlling roundoff error, sometimes by accepting some approximation error, is the main point of this chapter, and it will be a recurring theme throughout this book. J. H. Wilkinson [242] describes error analysis generally, with special emphasis on matrix methods. An excellent and comprehensive modern text is N. J. Higham's [104].

1.2 Floating point numbers

Binary computers represent everything in "bits" – fundamental units of measure that can take on the values 0 or 1 only. A byte is a collection of 8 bits, and real numbers are

typically stored in 4 bytes (single precision floating point) or 8 bytes (double precision floating point). The representation of real numbers in floating point format is governed by standards, the current one being IEEE Std. 754-2008 [1].

A decimal number like "0.1" means

$$0 \times 10^0 + 1 \times 10^{-1},$$

and in binary we can represent numbers in a similar fashion. The decimal number "23" has a binary representation

$$1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0,$$

or 10 111. The decimal number "0.1" has binary representation

$$1 \times 2^{-4} + 1 \times 2^{-5} + 0 \times 2^{-6} + 0 \times 2^{-7} + \cdots,$$

or

$$0.\overline{1100} \times 2^{-3}, \tag{1.1}$$

where the line indicates that the sequence 1100 repeats forever. There exist many numbers like "0.1" that can be represented in a compact fashion in decimal notation, but that cannot be represented in a compact notation in binary.

On a binary computer, the real number "0.1" is represented in single precision as

$$\underbrace{+}_{\text{sign}}\ 0.\underbrace{110\,011\,001\,100\,110\,011\,001\,101}_{\text{binary mantissa}}\ \underbrace{2^{-3}}_{\text{exponent}}, \tag{1.2}$$

with the 32 bits divided as follows:

- 1 sign bit
- 23+1 mantissa bits
- 8 exponent bits.

(The notation of (1.2) may be confusing because the exponent is clearly not written in a binary fashion. Of course it will be on a binary computer, but our concern here is with the mantissa.) The total number of bits in this example is 33, not 32. This is because we enforce (where possible) the following convention: the mantissa will be adjusted so that the first bit after the decimal place is 1. This is called a *normalized representation*. When this convention is respected, there is no need to store this bit – it is implicit.

In double precision, "0.1" is

$$+0.110\,011\,001\,100\,110\,011\,001\,100\,110\,011\,001\,100\,110\,011\,001\,100\,110\,10 \times 2^{-3}, \tag{1.3}$$

with

- 1 sign bit
- 52+1 mantissa bits
- 11 exponent bits.

Neither representation is exact, because the exact solution (1.1) cannot fit in 24 or 53 mantissa places. We will use the symbol t for the number of mantissa places, with $t = 53$ understood unless stated otherwise. Internal machine registers can have still more mantissa bits. For example, the Intel Pentium uses 64 mantissa bits in its internal floating point registers.

When a real number can be represented exactly with a machine floating point convention, it is called *machine representable*. If a real number is not machine representable, it is *rounded* to a close machine representable number. The rounding rule used by default on most modern desktop computers is called *round ties to even.* If the exact number lies exactly between two machine representable numbers, this rounding method chooses the machine representable number with 0 in the least significant bit. Otherwise, rounding chooses the closest machine representable number.

You can see in (1.3) that this rounding took place. The least significant bits of the exact mantissa read ...110 011... where the final underlined 1 is in the $2^{-(t+1)}$ place. By the rounding rule, the mantissa changed to ...110 10.

It is of interest to determine the maximum error that can be introduced by rounding. To determine this, consider the two binary numbers

$$x_1 = 0.011$$
$$x_2 = 0.100\overline{1} = 0.101.$$

x_1 is the smallest number that rounds up to 0.10, and x_2 is the largest number that rounds down to 0.10, with $t = 2$ and the round ties to even rule. These numbers both differ from 0.10 by 0.001, or $2^{-(t+1)}$, therefore rounding introduces a mantissa error as large as $2^{-(t+1)}$. Real numbers include also an exponent part, which we should account for. If a number x has a value of d in the exponent,

$$|x - \text{round}(x)| \leq 2^{-(t+1)} \times 2^d. \tag{1.4}$$

But, because of the normalized representation,

$$|x| \geq 0.1_2 \times 2^d = 2^{d-1} \tag{1.5}$$

(the subscript 2 was used to emphasize that 0.1 here is the leading part of the binary mantissa). Combining inequalities (1.4) and (1.5),

$$\frac{|x - \text{round}(x)|}{|x|} \leq 2^{-t} = \begin{cases} 2^{-24} \approx 6.0 \times 10^{-8} & \text{single precision} \\ 2^{-53} \approx 1.1 \times 10^{-16} & \text{double precision.} \end{cases}$$

The expression of this rounding error as a *relative error*, i.e., error divided by value, emphasizes the role of the finite mantissa.

Even if two numbers x and y were represented exactly in normalized binary form, their sum $x + y$ might not be. Likewise, for any function (multiplication, division, sine, cosine, exponentiation, etc.) the result of the operation – if carried out exactly – is unlikely to fit exactly into the finite number of mantissa places available for it. Let's denote $\tilde{x}$ as the rounded version of x, and $\tilde{f}$ the computed (hence rounded) version of

function $f(x, y)$. As a best case scenario, we have the situation

$$\begin{aligned} \tilde{f}(\tilde{x}, \tilde{y}) &= \text{round}(f(\tilde{x}, \tilde{y})) \\ \tilde{f}(\tilde{x}, \tilde{y}) - f(\tilde{x}, \tilde{y}) &= f(\tilde{x}, \tilde{y})\epsilon_f, \quad |\epsilon_f| \leq |2^{-t}|, \end{aligned} \tag{1.6}$$

which supposes that the function $f(\tilde{x}, \tilde{y})$ is as accurate as if it were computed exactly, but then rounded to fit the available space. We will make the optimistic assumption (1.6) that all *elementary operations* have an error equivalent to rounding. The number 2^{-t} is called *machine precision*, and will be denoted by symbol ϵ without subscripts.

If x and y are not machine representable, then, for example, $z = x + y$ has errors from the rounding of x, the rounding of y and the rounding of the sum. With z standing for the exact sum, and $\tilde{z}$ standing for the floating point version,

$$\begin{aligned} \tilde{z} &= \text{round}(\text{round}(x) + \text{round}(y)) \\ &= (x(1 + \epsilon_x) + y(1 + \epsilon_y))(1 + \epsilon_+) \quad \text{with all } |\epsilon_i| \leq \epsilon \equiv 2^{-t} \\ &\approx x + x(\epsilon_x + \epsilon_+) + y + y(\epsilon_y + \epsilon_+) \\ \frac{\tilde{z} - z}{z} &= \frac{\Delta z}{z} \approx \frac{x}{x + y}\epsilon_x + \frac{y}{x + y}\epsilon_y + \epsilon_+, \end{aligned} \tag{1.7}$$

where terms of *order of magnitude* ϵ^2 are ignored.

It is impossible to determine the error from (1.7) alone, because the relative errors ϵ_x, ϵ_y, and ϵ_+ are not known – we only know the upper bound of their magnitude. To use that information, we need to take the absolute value of (1.7) and use the triangle inequality:

$$\begin{aligned} \frac{|\Delta z|}{|z|} &= \left| \frac{x}{x + y}\epsilon_x + \frac{y}{x + y}\epsilon_y + \epsilon_+ \right| \\ &\leq \left| \frac{x}{x + y} \right| |\epsilon_x| + \left| \frac{y}{x + y} \right| |\epsilon_y| + |\epsilon_+| \\ &\leq \left(\left| \frac{x}{x + y} \right| + \left| \frac{y}{x + y} \right| + 1 \right) \epsilon. \end{aligned} \tag{1.8}$$

If $z = x + y$ is genuine addition (i.e., if the signs of x and y are identical), then $0 \leq x/(x + y) \leq 1$ and $0 \leq y/(x + y) \leq 1$, and $|x/(x + y)| + |y/(x + y)| = 1$ so the error has an upper bound of $|\Delta z/z| \leq 2\epsilon$.

However, if the signs of x and y are different, then this is really subtraction and $|x/(x + y)| \geq 1$. If z is a small difference in relatively large numbers then the factor $|x/(x + y)|$ can be very large, and the calculation $\tilde{z}$ can be correspondingly very inaccurate.

The reason for this is *cancellation*: when two numbers are subtracted, there can be a loss of significant figures. For example,

$$\begin{aligned} +0.100\,00_{10} &\approx +0.110\,011\,001\,100\,110\,011\,001\,101 \times 2^{-3} \\ -0.099\,85_{10} &\approx -0.110\,011\,000\,111\,111\,000\,101\,000 \times 2^{-3} \\ \hline +0.000\,15_{10} &\approx +0.100\,111\,010\,100\,100\,101\,010\,010 \times 2^{-12} = \text{round}(z), \end{aligned}$$

but

$$\begin{aligned}
&+0.110\,011\,001\,100\,110\,011\,001\,101 \times 2^{-3}\\
&-0.110\,011\,000\,111\,111\,000\,101\,000 \times 2^{-3}\\
&\text{---------------------------------}\\
&=0.\underline{000}\,\underline{000}\,\underline{000}\,100\,111\,010\,100\,101 \times 2^{-3}\\
&=0.100\,111\,010\,100\,101\,\underline{000}\,\underline{000}\,\underline{000} \times 2^{-12} = \tilde{z}.
\end{aligned}$$

Round(z) and $\tilde{z}$ differ in 10 places: we find a relative error of approximately 2^{-14} or about 1024ϵ. This difference is largely because upon subtraction 9 significant digits of the result were lost (underlined). When the normalized form is reestablished, these lost digits become zeros in the least significant figures.

Note that although there is a large loss of significant figures, this subtraction did not in itself introduce any numerical error [123, p. 12].† The effect of cancellation is to amplify the errors associated with rounding the inputs x and y.

A theoretical upper bound to the error from (1.8) is

$$\left(\frac{0.1}{0.000\,15} + \frac{0.099\,85}{0.000\,15} + 1\right)\epsilon \approx 1333\epsilon,$$

which is pretty close to the observed error.

The error formula (1.7) for addition has a special form that can be generalized to other functions. The idea is *differential error analysis*, and to understand it consider the function

$$y = \cos(x).$$

Accounting for roundoff error, the numerical result $\tilde{y}$ is

$$\tilde{y} = \text{round}(\cos(\tilde{x})) \tag{1.9}$$

and, in contrast to the expansion for addition, the exact value $y = \cos(x)$ does not appear when (1.9) is written

$$\tilde{y} = \cos(x(1+\epsilon_x))(1+\epsilon_c). \tag{1.10}$$

However, since ϵ_x is very small compared to 1, we could expand (1.10) in a Taylor series. Keeping only the leading terms,

$$\begin{aligned}
\tilde{y} &\approx \cos(x)(1+\epsilon_c) - x\epsilon_x \sin(x)\\
\frac{\Delta y}{y} &= \frac{\tilde{y}-y}{y} = -x\tan(x)\epsilon_x + \epsilon_c.
\end{aligned}$$

† "If p and q are represented exactly in the same conventional floating-point format, and if $1/2 \le p/q \le 2$, then $p-q$ too is representable exactly in the same format, unless $p-q$ suffers exponent underflow."

To generalize this to arbitrary unary and binary functions,

$$\frac{\Delta f(x)}{f(x)} = \left(\frac{x}{f}\frac{df}{dx}\right)\frac{\Delta x}{x} + \epsilon_f$$
$$\frac{\Delta g(x,y)}{g(x,y)} = \left(\frac{x}{g}\frac{dg}{dx}\right)\frac{\Delta x}{x} + \left(\frac{y}{g}\frac{dg}{dy}\right)\frac{\Delta y}{y} + \epsilon_g.$$

The factors

$$\left(\frac{x}{f}\frac{df}{dx}\right)$$

are called *condition numbers* or *amplification factors*. For genuine addition, we saw that these numbers lie between 0 and 1, so they do not cause error to grow. For subtraction, these numbers are greater than 1 in magnitude and they amplify error.

1.3 Algorithms and error

Formulas that are mathematically identical can incur different numerical errors, therefore to assess the numerical error associated with some function it is important to specify completely the algorithm that will be used to evaluate it.

Example 1.1 The variance S^2 of a set of observations $x_1, ..., x_n$ is to be determined (S is the standard deviation). Which of the formulas,

$$S_1^2 = \frac{1}{n-1}\left(\sum_{i=1}^{n} x_i^2 - n\bar{x}^2\right) \tag{1.11a}$$
$$S_2^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2, \tag{1.11b}$$

with

$$\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i,$$

is better from a numerical error point of view? We would like to ask "which formula has the lowest error," but this question cannot be answered. Instead, we can ask and answer "which formula has the lowest maximum error?" This is sometimes worded "which algorithm is more *numerically trustworthy*?"

Solution

First, we note that these expressions are mathematically identical:

$$\begin{aligned} S_2^2 &= \frac{1}{n-1}\sum_{i=1}^{n}(x_i-\bar{x})^2 \\ &= \frac{1}{n-1}\left[\sum_{i=1}^{n}x_i^2 - 2\sum_{i=1}^{n}x_i\bar{x} + \sum_{i=1}^{n}\bar{x}^2\right] \\ &= \frac{1}{n-1}\left[\sum_{i=1}^{n}x_i^2 - 2n\bar{x}^2 + n\bar{x}^2\right] \\ &= \frac{1}{n-1}\left[\sum_{i=1}^{n}x_i^2 - n\bar{x}^2\right] = S_1^2, \end{aligned}$$

so any numerical differences will be due to the algorithm only.

To answer the question of numerical trustworthiness, we need to formally state the algorithm associated with the mathematical formulas, then analyze these algorithms by tracking the rounding errors.

The first algorithm can be written, for $n = 2$, $\phi_1 = x_1 + x_2$, $\bar{x} = \phi_1/2$, $\phi_2 = x_1^2$, $\phi_3 = x_2^2$, $\phi_4 = \phi_2 + \phi_3$, $\phi_5 = \bar{x}^2$, $\phi_6 = n\phi_5$, $\phi_7 = \phi_4 - \phi_6$, $S_1^2 = \phi_7/(n-1)$. The error associated with this algorithm might be written with differential error analysis

$$\begin{aligned} \frac{\Delta x_1}{x_1} &= \epsilon_{x_1} \\ \frac{\Delta x_2}{x_2} &= \epsilon_{x_2} \\ \frac{\Delta\phi_1}{\phi_1} &= \frac{x_1}{\phi_1}\frac{\Delta x_1}{x_1} + \frac{x_2}{\phi_1}\frac{\Delta x_2}{x_2} + \epsilon_1 \\ \frac{\Delta\bar{x}}{\bar{x}} &= \frac{\Delta\phi_1}{\phi_1} \\ \frac{\Delta\phi_2}{\phi_2} &= 2\frac{\Delta x_1}{x_1} + \epsilon_2 \\ \frac{\Delta\phi_3}{\phi_3} &= 2\frac{\Delta x_2}{x_2} + \epsilon_3 \\ \frac{\Delta\phi_4}{\phi_4} &= \frac{\phi_2}{\phi_4}\frac{\Delta\phi_2}{\phi_2} + \frac{\phi_3}{\phi_4}\frac{\Delta\phi_3}{\phi_3} + \epsilon_4 \\ \frac{\Delta\phi_5}{\phi_5} &= 2\frac{\Delta\bar{x}}{\bar{x}} + \epsilon_5 \\ \frac{\Delta\phi_6}{\phi_6} &= \frac{\Delta\phi_5}{\phi_5} \\ \frac{\Delta\phi_7}{\phi_7} &= \frac{\phi_4}{\phi_7}\frac{\Delta\phi_4}{\phi_4} - \frac{\phi_6}{\phi_7}\frac{\Delta\phi_7}{\phi_7} + \epsilon_7 \\ \frac{\Delta S_1^2}{S_1^2} &= \frac{\Delta\phi_7}{\phi_7}. \end{aligned} \tag{1.12}$$

Note that division by 1 carries no error, and division by 2 carries no error either – division or multiplication by any power of 2 will affect the exponent of the number, not its mantissa.

Combining the expressions, and writing the intermediate variables in terms of x_1 and x_2 gives:

$$\begin{aligned}\frac{\Delta S_1^2}{S_1^2} = &\underbrace{\epsilon_7 + \frac{2x_1}{x_1 - x_2}\epsilon_{x_1} - \frac{2x_2}{x_1 - x_2}\epsilon_{x_2}} \\ &- 2\frac{(x_1 + x_2)^2}{(x_1 - x_2)^2}\epsilon_1 + 2\frac{x_1^2}{(x_1 - x_2)^2}\epsilon_2 \\ &+ 2\frac{x_2^2}{(x_1 - x_2)^2}\epsilon_3 + 2\frac{x_1^2 + x_2^2}{(x_1 - x_2)^2}\epsilon_4 - \frac{(x_1 + x_2)^2}{(x_1 - x_2)^2}\epsilon_5.\end{aligned} \tag{1.13}$$

The collection of terms with a bracket beneath them is special, as will be explained.

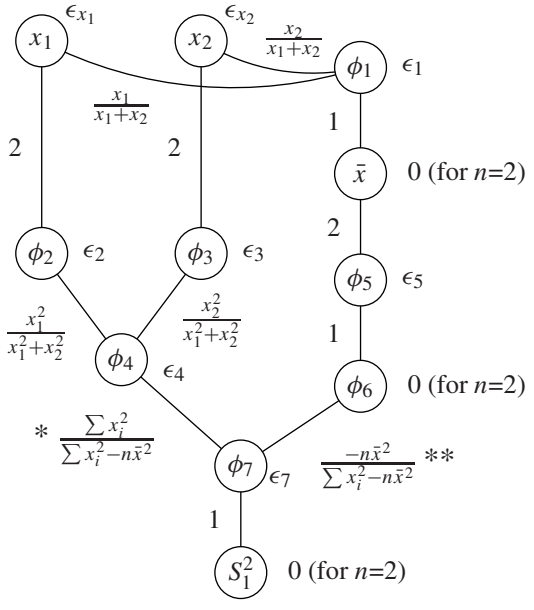

Figure 1.1 Bauer graph for S_1^2 with $n = 2$; Example 1.1.

Finally, before analyzing ΔS_2^2, note that there is a convenient graphical method to obtain the information embodied in (1.12). A "Bauer graph" [9] of algorithm 1 is given in Figure 1.1. A circle is drawn for each input variable and for each *elementary operation*, and lines are drawn to show how the data from one circle connects to the others. Associated with each circle is a rounding error, and associated with each line of the graph is the condition number of the elementary operation. The graph is "read" by tracing all paths to the final result, multiplying each rounding error by the product of condition numbers that follow it.

The second algorithm may be written: $n = 2$ and $\eta_1 = x_1 + x_2$, $\bar{x} = \eta_1/2$, $\eta_2 = x_1 - \bar{x}$, $\eta_3 = \eta_2^2$, $\eta_4 = x_2 - \bar{x}$, $\eta_5 = \eta_4^2$ $\eta_6 = \eta_4 + \eta_5$, $S_2^2 = \phi_6/(n-1)$. To analyze this algorithm we'll use its Bauer graph (Figure 1.2).

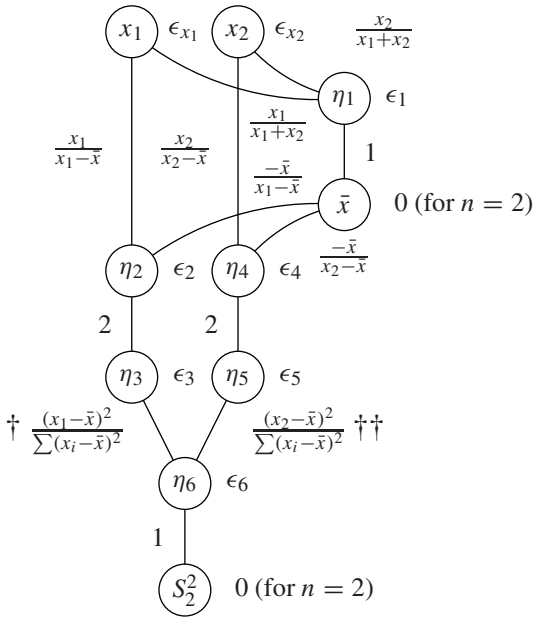

Figure 1.2 Bauer graph for S_2^2 with $n = 2$; Example 1.1.

Analyzing the graph gives the same result that would be obtained if each differential error expansion were written and combined:

$$\frac{\Delta S_2^2}{S_2^2} = \underbrace{\epsilon_6 + \frac{2x_1}{x_1 - x_2}\epsilon_{x_1} - \frac{2x_2}{x_1 - x_2}\epsilon_{x_2}} + \epsilon_2 + \frac{1}{2}\epsilon_3 + \epsilon_4 + \frac{1}{2}\epsilon_5. \tag{1.14}$$

It is interesting to note that there is no dependence here at all on the relative error ϵ_1.

A collection of terms has been identified with a bracket in (1.13) and in (1.14), and these have the same bound. These bracketed errors are (1) the input errors ϵ_{x_1} and ϵ_{x_2}, propagated through to the result S^2, and (2) one roundoff error associated with the last step of the algorithm. This final error is included because, like (1.6), even an exact calculation will be subject to rounding. Any mathematically identical algorithm for calculating S^2 will contain these terms – they are completely unavoidable. They are called *inherent error*, to distinguish them from the algorithm-dependent errors that make up the remainder of (1.13) and (1.14).

We have, and will continue to assume in general, that $|\epsilon_i| \leq \epsilon = 2^{-t}$ for all errors in the calculation. But, in fact, input error ϵ_x may be considerably worse than ϵ because of experimental errors. The inherent error may be used to analyze how experimental error affects the calculated solution.

Now, we compare the algorithm-dependent errors in (1.13) and (1.14). Algorithm 1 incurs large cancellation errors at the end of the algorithm: condition numbers (*) and (**) of Figure 1.1 are guaranteed ≥ 1 and ≤ -1, respectively, because actual subtraction occurs. No such large condition numbers appear in the algorithm-dependent part of the error for algorithm 2. In algorithm 2 the final steps are *numerically harmless*, and the amplification factors (†) and (††) of Figure 1.2 lie between 0 and 1. This results in a numerically stable algorithm because the roundoff errors grow like $|x_1 - x_2|^0$ while the inherent errors grow like $|x_1 - x_2|^{-1}$. Algorithm 2 is more numerically trustworthy than algorithm 1.

The lesson of this example is that when cancellation is unavoidable, do it as soon as possible. If it is postponed, the large condition numbers amplify numerical errors beyond those occurring in the inherent error.

To demonstrate the difference in these algorithms, the calculation is performed on the following two columns of numbers. Both columns have the same variance.

data set 1	data set 2
-0.329	$10^6 - 0.329$
0.582	$10^6 + 0.582$
-0.039	$10^6 - 0.039$
-0.156	$10^6 - 0.156$
-0.063	$10^6 - 0.063$
0.552	$10^6 + 0.552$
-0.927	$10^6 - 0.927$
-0.540	$10^6 - 0.540$
0.460	$10^6 + 0.460$
0.046	$10^6 + 0.046$

set		algorithm 1	algorithm 2
1	S^2	`2.3655560000000003e-01`	`2.3655560000000003e-01`
	ΔS^2	`1.1733214354523381e-16`	`1.1733214354523381e-16`
2	S^2	`2.3676215277777779e-01`	`8.7316799001074320e-04`
	ΔS^2	`2.3655560001559353e-01`	`6.5919075558009081e-11`

For the first set of data both algorithms compute the solution to approximately machine precision. With the second data set the differences are dramatic. The relative error of algorithm 1 is $\approx 10^7$ times worse than the relative error of algorithm 2.

Example 1.2 Consider two algorithms for the same function

$$y_1 = \frac{1 - \cos(x)}{x}$$

$$y_2 = \frac{\sin^2(x)}{x\,(1 + \cos(x))},$$