§5 Errors and Uncertainties in Computation

Whether you are careful or not, errors and uncertainties are an inevitable part of computation. Some errors are the ones that humans unavoidably make, but some are introduced by the computer. Computer errors arise due to the limited precision with which computers store numbers or the failure of algorithms or models. Although it suppresses creativity to keep thinking "error" when approaching a computation, it is certainly a waste of time and may lead to harm to work with results that are meaningless due to errors. In this chapter, we examine the errors and uncertainties that may occur in computations.

5.1 Types of Errors

To investigate why errors in a program should be of concern, consider a program with the logical flow

$$\text{start} \to U_1 \to U_2 \to \ldots \to U_n \to \text{end}$$

where each unit U may be a statement or step. If each unit has probability p of being correct, then the joint probability P of the whole program being correct is $P = p^n$. Suppose we have a medium-sized program with $n = 1000$ steps and that the probability of each step being correct is almost one, say $p = 0.9993$. Then you end up with $P \approx 0.5$, i. e. a final answer that is as likely wrong as right. The problem is that, as a scientist, you want a result that is correct — or at least in which the uncertainty is small and of known size.

In general, there are four types of errors that may exist in your computations:

1. **Blunders or bad theory**: typographical errors entered with your program or data, running the wrong program or having a fault in your theory, using the wrong data file, and so on.
2. **Random errors**: imprecision caused by events such as fluctuations in electronics, cosmic rays, or someone pulling a plug. These may be rare, but you have no control over them and their likelihood increases with running time. You may have confidence that these errors will not have much effect on a 10-s calculation. However, a week-long calculation may have to be run several times to ensure reproducibility.
3. **Truncation errors**: imprecision arising from simplifying the mathematics in order to solve a problem on the computer. They include the replacement of infinite series by finite sums, infinitesimal intervals by finite ones, and variable functions by constants.

For example,

$$\sin x = \sum_{i=0}^{\infty} \frac{(-1)^{i-1} x^{2i-1}}{(2i-1)!} \quad \text{(exact)}$$
$$\approx \sum_{i=0}^{N} \frac{(-1)^{i-1} x^{2i-1}}{(2i-1)!} = \sin x + \xi(x, N), \quad \text{(algorithm)}$$

where $\xi(x, N)$ is the truncation error and it is the series from $N+1$ to ∞ in this case. Because truncation error arises from the algorithm we use to approximate the mathematics, it is also called **algorithmic error** or **approximation error**. For every reasonable approximation, the truncation error should decrease as N increases and vanish in the $N \to \infty$ limit. In the above example, because the scale for N is set by the value of x, a small truncation error requires $N \gg x$. Thus, the truncation error will be large if x and N are close in value.

4. **Round-off errors**: imprecision arising from the finite number of digits used to store floating-point numbers. These errors are analogous to the uncertainty in the measurement of a physical quantity encountered in a physics laboratory. The overall round-off error accumulates as the computer handles more numbers, i. e. as the number of steps in a computation increases, and may cause some algorithms to become unstable with a rapid increase in error. In some cases, round-off error may become the major component in your answer, leading to what computer experts call **garbage**. For example, if your computer kept only four decimal places, then it will store 1/7 as 0.1429 and 2/7 as 0.2857, where the computer has rounded off the last digit in 1/7. Thus, if we ask the computer to do a simple calculation $2(1/7) - 2/7$, it produces

$$2\left(\frac{1}{7}\right) - \frac{2}{7} = 0.2858 - 0.2857 = 0.0001 \neq 0.$$

So the result is not 0 although it is small. If we repeat this type of calculation millions times, then the final answer might not even be small (garbage begets garbage).

Here we will focus on the study of truncation errors and round-off errors in computation as these errors can be reduced by using an appropriate algorithm.

5.2 Representation of Numbers in Computer

To analyze errors in computation, we need to understand how numbers are stored in computer since it would affect the precision of calculations. The most elementary units of computer memory are the **bits** which store binary integers 0 and 1. Thus all numbers

are stored in memory in **binary** form, i. e. as long strings of zeros and ones. As a result, N bits can store integers in the range $[0, 2^N - 1]$; however, because the sign of the integer is represented by the first bit (a zero bit for positive numbers), the actual range decreases to $[0, 2^{N-1} - 1]$.

Long strings of zeros and ones are fine for computers but are awkward for users. Thus binary strings are converted to octal, decimal, or hexadecimal numbers before the results are communicated to people. Octal and hexadecimal numbers are nice since the conversion loses no precision, but not all that nice because our decimal rules of arithmetic don't work for them. Converting to decimal numbers makes the numbers easier for us to work with, but it leads to a decrease in precision unless the number is a power of 2.

A description of a particular computer system normally states the **word length**, i. e. the number of bits used to store a number. The length is often expressed in **bytes**, with 1 byte equal to 8 bits. Memory and storage sizes are measured in bytes (B), kilobytes (kB), megabytes (MB), gigabytes (GB), and terabytes (TB). Some care should be taken here for those who chose to compute sizes in detail because K does not always mean 1000. For instance, 1 kilobyte is often used to refer to 1024 bytes (2^{10} bytes).

The memory chips in some older personal computers used 8-bit words. It means that the maximum integer was $2^7 - 1 = 127$ (7 because 1 bit is used for the sign). On the other hand, using 64 bits permits integers in the range up to $2^{63} - 1 \approx 10^{19}$. This may seem like a large range at the first glance. But it is not really when compared to the range of numbers encountered in the physical world.

Most scientific computations use double precision floating-point numbers which store numbers in 8 bytes. The floating-point representation of numbers on computers is a binary version of **scientific notation** or **engineering notation**. For example, the speed of light c is $+2.99792458{\times}10^{+8}$ m/s in scientific notation and $+0.299792458{\times}10^{+9}$ m/s in engineering notation. In each of these cases, the number in front is called the **mantissa** and contains nine significant figures. The power to which 10 is raised is called the **exponent**, with the plus sign included as a reminder that these numbers may be negative.

Floating-point numbers are stored on the computer as a concatenation of the sign bit, the exponent, and the mantissa. Because only a finite number of bits are stored, the **machine numbers**, which is the set of floating-point numbers that the computer can store exactly, is much smaller than the set of real numbers. In particular, machine numbers have a maximum and a minimum. An error condition known as **overflow** occurs

if you exceed the maximum while an error condition known as **underflow** occurs if you fall below the minimum. In the latter case, the software and hardware may be set up so that underflows are set to zero without you even being told. In contrast, overflows usually stop the execution of the computation.

The actual relation between what is stored in memory and the value of a floating-point number is somewhat indirect, with there being a number of special cases and relations used over the years. In fact, each computer operating system and each computer language adopted its own standards for floating-point numbers in the past. Different standards meant that the same program running correctly on different computers could give different results. Even though the results usually were only slightly different, the user could never be sure if the lack of reproducibility of a test case was due to the particular computer being used or an error in the program's implementation.

Table 5.1: The IEEE 754 Standard for Primitive Data Types

Name	**Type**	**Bits**	**Range**
`boolean`	Logical	1	`true` or `false`
`char`	String	16	'\u0000' to '\uFFFF' (ISO Unicode characters)
`byte`	Integer	8	-128 to $+127$
`short`	Integer	16	-32768 to $+32767$
`int`	Integer	32	-2147483648 to $+2147483647$
`long`	Integer	64	-9223372036854775808 to $+9223372036854775807$
`float`	Floating	32	$\pm 1.401298 \times 10^{-45}$ to $\pm 3.402823 \times 10^{+38}$
`double`	Floating	64	$\pm 4.9406564584124654 \times 10^{-324}$ to $\pm 1.7976931348623157 \times 10^{+308}$

In 1987, the Institute of Electrical and Electronics Engineers (IEEE) and the American National Standards Institute (ANSI) adopted the IEEE 754 standard for floating-point arithmetic. When the standard is followed, you can expect the primitive data types to have the precision and ranges given in Table 5.1. In addition, when computers and software adhere to this standard, you are guaranteed that your program will produce identical results on different computers. However, because the IEEE standard may not produce the most efficient code or the highest accuracy for a particular computer, sometimes you may have to invoke compiler options to demand that the IEEE standard be strictly followed for your test cases. After you know that the code is okay, you may want to run with others that gives the greatest speed and precision.

In the IEEE representations, a floating-point number x is normally stored as

$$x = (-1)^s \times 1.f \times 2^{e-b},$$

i. e. with separate entities for the sign s, the fractional part of the mantissa f, and the exponent field e. Floating numbers stored in this way are known as **normal numbers**. Note that b is a fixed number called **bias** that is added to the actual exponent p before it is stored as the biased exponent e. So the actual exponent is $p = e - b$ which may be negative. The IEEE 754 Standard supports two basic floating-point formats: **single-precision floating point numbers** (singles) and **double-precision floating point numbers** (doubles). Singles occupy 32 bits overall, with 1 bit for the sign s, 8 bits for the exponent e, and 23 bits for the fractional mantissa f. Doubles occupy 64 bits overall, with 1 bit for the sign s, 11 bits for the exponent e, and 52 bits for the fractional mantissa f. Moreover, the bias is 127 for singles and 1023 for doubles. Notice that in the case the biased exponent e is zero but the mantissa f is non-zero, a floating point number is stored as

$$x = (-1)^s \times 0.f \times 2^{1-b},$$

Floating numbers stored in this way are known as **subnormal numbers**. On the other hand, a signed zero is represented by a floating point number with both zero biased exponent e and zero mantissa f.

Note that the values $\pm$`INF` and `NaN` are not numbers in the mathematical sense, i. e. objects that can be manipulated or used in calculations to take limits and so on. Indeed, they are signals to the computer and to you that something has gone awry and that the calculation should probably stop until you straighten things out. These values are represented by special floating point numbers in the IEEE representations.

Python is a relatively new language with changes and extensions occurring as its use spreads and as its features mature. It should be no surprise that Python does not adhere to all aspects currently, and especially the special cases of the IEEE 754 standard. Probably the most relevant difference for us is that Python does not support single-precision floating point numbers. A data type `float` in Python is the equivalent of a `double` in the IEEE standard. So in Python the round-off error for the calculation of a number $f(x)$ is approximately $Cf(x)$ where C is typically about 10^{-16}. This is not a loss since singles are inadequate for most scientific computing. However, if you switch over to Java or C, you should declare your variables as `double` and not as `float`. While Python eliminates single-precision floating point numbers, it adds a new data type `complex` for dealing with complex numbers.

The details of how closely Python adheres to the IEEE 754 standard depend upon the details of Python's use of the C or Java language to power the Python interpreter. In particular, with the recent 64 bit architectures for CPUs, the range may even be greater than the IEEE standard, and the abnormal numbers (±`INF` and `NaN`) may differ. Similarly, the exact conditions for overflows and underflows may also differ.

5.3 Round-off Error in Computation

It's very often that numbers are stored only approximately on the computer. A calculation employing these numbers can be expected to yield only an approximate answer. To demonstrate the effect of this type of uncertainty, we model the computer representation x_c of the exact number x as

$$x_c \approx x(1+\epsilon_x), \tag{5.1}$$

where ϵ_x is the relative error in x_c that is expected to be of a similar magnitude to the machine precision ϵ_m (which is about 10^{-16} in Python). If we apply the notation in Eq. (5.1) to the subtraction $a = b - c$, then we obtain

$$\begin{aligned} a_c = b_c - c_c &\approx b(1+\epsilon_b) - c(1+\epsilon_c) \\ \Rightarrow \quad \frac{a_c}{a} &\approx 1 + \epsilon_b\frac{b}{a} - \epsilon_c\frac{c}{a} \end{aligned} \tag{5.2}$$

From Equation (5.2), we can see that the resulting round-off error in a is actually a weighted average of the errors in b and c, with no assurance that the last two terms will cancel out each other. Of special importance here is to observe that the error in the answer a_c increases when we subtract two nearly equal numbers ($b \approx c$) because then we are subtracting off the most significant parts of both numbers and leaving the error-prone least-significant parts:

$$\begin{aligned} \frac{a_c}{a} &\stackrel{\text{def}}{=} 1 + \epsilon_a \approx 1 + \frac{b}{a}(\epsilon_b - \epsilon_c) \\ \Rightarrow \quad \frac{a_c}{a} &\approx 1 + \frac{b}{a}\max(|\epsilon_b|, |\epsilon_c|) \end{aligned} \tag{5.3}$$

This shows that even if the relative errors in b and c may cancel somewhat, they are multiplied by the large number b/a, which can significantly magnify the error. Because we cannot assume any sign for the errors, we must assume the worst case (i. e. the "max" in Eq. (5.3)). Such observation leads us to the following theorem:

If you subtract two large numbers and end up with a small one, then there will be less significance, and possibly a lot less significance, in the small one.

We have already seen an example of **subtractive cancellation** in the power series summation for $\sin x \approx x - x^3/3! + x^5/5! + \ldots$ for large x. A similar effect occurs $e^{-x} \approx 1 - x + x^2/2 - x^3/3 + \ldots$ for large x, where the first few terms are large but of alternating sign, leading to an almost total cancellation in order to yield the final small result. (Subtractive cancellation can be eliminated by using the identity $e^{-x} = 1/e^x$, although round-off error will still remain.)

Next, we consider how round-off error arises from a single division of the computer representations of two numbers. We apply the notation in Eq. (5.1) to the division $a = b/c$:

$$a_c = \frac{b_c}{c_c} \approx \frac{b(1+\epsilon_b)}{c(1+\epsilon_c)}$$

$$\Rightarrow \quad \frac{a_c}{a} \approx \frac{1+\epsilon_b}{1+\epsilon_c} \approx (1+\epsilon_b)(1-\epsilon_c) \approx 1 + \epsilon_b - \epsilon_c$$

$$\Rightarrow \quad \frac{a_c}{a} \approx 1 + |\epsilon_b| + |\epsilon_c| \tag{5.4}$$

Here we ignore the very small ϵ^2 terms and add errors in absolute value since we cannot assure that the unknown errors cancel out each other. Since we add the errors in absolute value, this same rule holds for multiplication. Equation (5.4) is just the basic rule of error propagation from elementary laboratory work. That is to say, you add the uncertainties in each quantity involved in an analysis to arrive at the overall uncertainty.

We can even generalize this model to estimate the error ξ in the evaluation of a general function $f(x)$, i. e. the difference in the value of the function evaluated at x and at x_c:

$$\xi = \frac{f(x) - f(x_c)}{f(x)} \approx \frac{f'(x)}{f(x)}(x - x_c)$$

For example, if $f(x) = (1+x)^{1/3}$, then

$$\frac{df}{dx} = \frac{1}{3(1+x)^{2/3}}$$

$$\Rightarrow \quad \xi \approx \frac{1}{3(1+x)}(x - x_c)$$

If we evaluate this expression for $x = \pi/4$ and assume an error in the fourth place of x, we obtain a relative error of 1.9×10^{-5} in $(1+x)^{1/3}$.

There is a useful model for approximating how round-off error accumulates in a calculation involving a large number of steps. We view the error in each step as a literal "step" in a random walk, i. e. a walk for which each step is in a random direction. It can be shown that the average total distance covered in N steps of length r is

$$R \approx \sqrt{N} r$$

By analogy, the average total relative error $\epsilon_{\rm ro}$ arising after N computational steps each with machine precision error ϵ_m is

$$\epsilon_{\rm ro} \approx \sqrt{N}\epsilon_m \tag{5.5}$$

If the round-off errors in a particular algorithm do not accumulate in a random manner, then a detailed analysis is needed to predict the dependence of the error on the number of steps N. In some cases there may be no cancellation, and the error may increase as $N\epsilon_m$. Even worse, the error increases as $N!$ in some recursive algorithms where the error generation is coherent.

Our discussion of errors has an important implication for a student to keep in mind when they perform calculation requiring hours of supercomputer time. A fast computer may complete 10^{10} floating-point operations per second. It means that a program running for 3 hours performs approximately 10^{14} operations. Thus, if round-off error accumulates randomly, then we expect a relative error of $10^7\epsilon_m$ after 3 hours in accordance with Eq. (5.5). For the error to be smaller than the answer, we need $\epsilon_m < 10^{-7}$, which requires double precision and a good algorithm. If we want a higher-precision answer, then we will need a very good algorithm.

5.4 Propagation of Errors in Numerical Algorithms

Numerical algorithms play a vital role in computational physics. Your problem is to take a general algorithm and decide

1. Does it converge to a limit?
2. How precise are the converged results?
3. How time-consuming is it?

At first glance, these problems seems dumb. It's because all algorithms converge if enough terms are used, and more precise result would be obtained if more terms are used. Well, some algorithms may be asymptotic expansions that just approximate a function in certain regions of parameter space and converge only up to a point. But even if a uniformly convergent power series is used as the algorithm, including more terms will decrease the truncation error but increase the round-off error. And because round-off error eventually diverges to infinity, the best we can hope for is a "best" approximation. Good algorithms are good not only because they are fast but also because being fast means that round-off error does not have much time to grow.

Let us assume that an algorithm takes N steps to find a good answer. As a rule of thumb, the relative truncation error ϵ_{trun} decreases rapidly, often as the inverse power of the number of terms used:

$$\epsilon_{\text{trun}} \approx \frac{\alpha}{N^{\beta}} \tag{5.6}$$

Here α and β are empirical constants that change for different algorithms and may be only approximately constant, and even then $\epsilon_{\text{trun}} \to 0$ only as $N \to \infty$. The fact that the error must fall off for large N is just a statement that the algorithm converges.

In contrast to this truncation error, round-off error tends to grow slowly and somewhat randomly with N. If the round-off errors in each step of the algorithm are not correlated, then we know from our previous discussion that the accumulation of error can be modeled as a random walk with step size equal to the machine precision ϵ_m. So the relative round-off error is given by $\epsilon_{\text{ro}} \approx \sqrt{N}\epsilon_m$ as in Eq. (5.5). This is the slow growth with N that we expect from the round-off error. The total relative error in a computation is the sum of these two types of errors:

$$\epsilon_{\text{tot}} = \epsilon_{\text{trun}} + \epsilon_{\text{ro}} \approx \frac{\alpha}{N^{\beta}} + \sqrt{N}\epsilon_m \tag{5.7}$$

For small N, we expect the first term to be the larger of the two but ultimately to be overcome by the slowly growing round-off error.

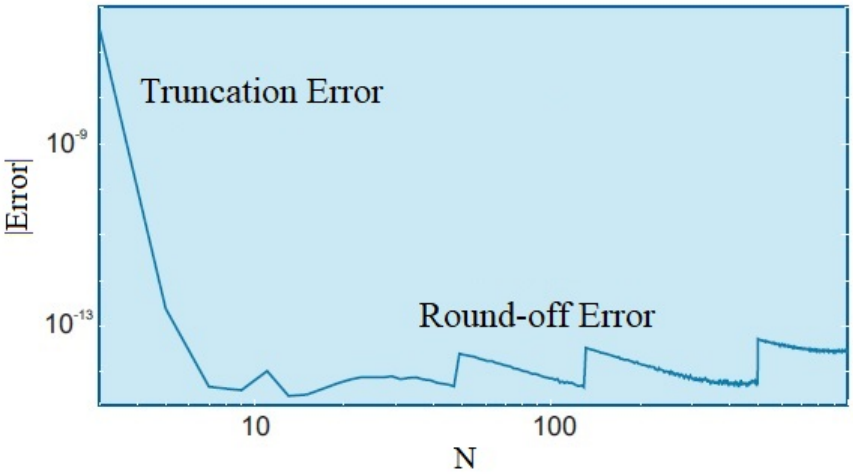

Figure 5.1: A log-log plot of relative error versus the number of points N used in a numerical integration using the Simpson's rule.

As an example, Figure 5.1 shows a log-log plot of the relative error in a numerical integration using the Simpson rule (which is discussed in Chapter 6). We consider the $\log_{10}$ of the relative error because its negative tells us the number of decimal places of

precision obtained. For instance, if we assume $\mathcal{A}$ to be the exact answer and $A(N)$ to be the computed answer, then

$$\frac{\mathcal{A} - A(N)}{\mathcal{A}} \approx 10^{-9}$$

$$\Rightarrow \log_{10} \left| \frac{\mathcal{A} - A(N)}{\mathcal{A}} \right| \approx -9$$

From Figure 5.1, we observe that the error shows a rapid decrease for small N which is consistent with the inverse power law in Eq. (5.6). In this region, the algorithm is converging. As N is increased, the error starts to look somewhat erratic, with a slow increase on the average. In accordance with Eq. (5.7), the round-off error has grown larger than the truncation error in this region and will continue to grow for increasing N. Obviously, the smallest total relative error will then be obtained if we stop the calculation at the minimum near 10^{-14}, i. e. when $\epsilon_{\text{trun}} \approx \epsilon_{\text{ro}}$.

You would not know the exact answer in realistic calculations. However, you may know the exact answer for a similar calculation. And you can use that similar calculation to perfect your numerical technique. Alternatively, as you understand the behavior of the total error in a computation, you should be able to deduce the manner in which your algorithm is converging from a table or a graph of your answer. In particular, at some point you should see that the mantissa of the answer changes only in the less significant digits with that place moving further to the right of the decimal point as more steps are executed in the calculation. However, as the number of steps becomes even larger, the round-off error eventually leads to a fluctuation in the less significant digits, with a gradual increase on the average. It is the best to quit the calculation before this occurs.

Based upon this understanding, an approach to obtain the best approximation is to deduce when your answer behaves like Eq. (5.7). To do that, we call $\mathcal{A}$ the exact answer and $A(N)$ the computed answer after N steps. Moreover, we assume that for large enough values of N, the approximation converges as

$$A(N) \approx \mathcal{A} + \frac{\alpha}{N^{\beta}} \mathcal{A}$$

In other words, the round-off error term in Eq. (5.7) is still small. We then run our computer program with $2N$ steps, which should give a better answer, and use that answer to eliminate the unknown $\mathcal{A}$:

$$A(N) - A(2N) \approx \frac{\alpha}{N^{\beta}} \mathcal{A}$$

To see if these assumptions are correct and determine what level of precision is possible for the best choice of N, plot $\log_{10} |[A(N) - A(2N)]/A(2N)|$ versus $\log_{10} N$, similar to what

we have done in Figure 5.1. If you obtain a rapid straight-line drop off, then you know that you are in the region of convergence and can deduce a value for β from the slope of the line. As N becomes larger, you should see the graph change from a straight-line decrease to a slow increase as the round-off error begins to dominate. A good place to quit is before this occurs. In any case, now you understand the error in your computation and thus have a chance to control it.

To illustrate how different kinds of errors enter into a computation, we assume that the analytic form for the relative truncation and round-off errors of an algorithm are:

$$\epsilon_{\text{trun}} \approx \frac{1}{N^2}, \quad \epsilon_{\text{ro}} \approx \sqrt{N}\epsilon_m$$

$$\Rightarrow \ \epsilon_{\text{tot}} = \epsilon_{\text{trun}} + \epsilon_{\text{ro}} \approx \frac{1}{N^2} + \sqrt{N}\epsilon_m$$

Then the total relative error reaches the minimum value when

$$\frac{d\epsilon_{tot}}{dN} \approx -\frac{2}{N^3} + \frac{\epsilon_m}{2\sqrt{N}} = 0$$

$$\Rightarrow \ N \approx \left(\frac{4}{\epsilon_m}\right)^{2/5}$$

For a single-precision calculation with $\epsilon_m \approx 10^{-7}$, the minimum total relative error occurs when

$$N \approx \left(\frac{4}{10^{-7}}\right)^{2/5} \approx 1099 \qquad \Rightarrow \ \epsilon_{\text{tot}} \approx 4\times 10^{-6}$$

In this case, most of the error is due to round-off error and not truncation error. Also observe that even for this case, the best we can do is about 40 times machine precision.

Note that the total relative error is mainly the round-off error $\propto \sqrt{N}$ when it attains the minimum value. So an obvious way to decrease this error is to use a smaller number of steps N. Let us assume that we do this by finding another algorithm that converges more rapidly with N, say, one with relative truncation error behaving like

$$\epsilon_{\text{trun}} \approx \frac{2}{N^4}$$

Then the total relative error is

$$\epsilon_{\text{tot}} = \epsilon_{\text{trun}} + \epsilon_{\text{ro}} \approx \frac{2}{N^4} + \sqrt{N}\epsilon_m$$

The number of points for the minimum total relative error is found like that before:

$$\frac{d\epsilon_{tot}}{dN} = 0 \qquad \Rightarrow \ N \approx \left(\frac{16}{\epsilon_m}\right)^{2/9}$$

Therefore, for single-precision calculation with $\epsilon_m \approx 10^{-7}$,

$$N \approx \left(\frac{16}{10^{-7}}\right)^{2/9} \approx 67 \quad \Rightarrow \ \epsilon_{\text{tot}} \approx 9 \times 10^{-7}$$

The error is now smaller by a factor of 4, with only 1/16 as many steps needed. Here we can observe a subtle behavior of the computer. In this case the better algorithm is quicker and produces less round-off error by using fewer steps.