Reprint from
The *Mathematica* Conference

June, 1992   Boston, MA

831.21.10.1992

Numerical Computation II

Jerry Keiper

1 Controlling the Effects of Numerical Errors

A common misconception regarding errors in numerical computation is that the error in the final result is the simple accumulation of the errors made in the intermediate results. It is certainly true that if you reduce the errors in the intermediate results the error in the final result will (usually) also be reduced. By using high-precision arithmetic you can generally make the error in the final result arbitrarily small, but such an approach to error reduction can be very much more expensive than is necessary.

The first thing to realize about error in computer arithmetic is that it is not something that is absolutely unavoidable. Computer arithmetic is deliberately designed to involve errors and to have the errors behave in certain well defined ways. The reason we have errors is that compromises are made to achieve efficiency both in terms of speed and memory usage. With respect to basic arithmetic *Mathematica* provides both integers and rational numbers and there is no error involved using them. Of course if you want to find the sum

$$\sum_{i=1}^{\infty} 2^{-i}$$

and you attempt to evaluate it using rational numbers you will get into trouble. If it weren't for the fact that you would run out of memory it would literally take forever. So, while you can often avoid errors, you can't always get useful answers by doing so.

For practical purposes errors are thus inevitable except in rather trivial problems. They arise from several sources: data error, truncation error, and representation error. By data error we mean any error in the input data to a problem or a subproblem. This includes all the errors in the results of prior subproblems even if their ultimate source is truncation error or representation error. By truncation error we mean primarily errors resulting from approximation of infinite processes by finite processes, *e.g.*, the approximation of a definite integral by a finite sum. Representation error is error that is intentionally made to allow values that cannot be represented exactly to be represented approximately. The most common type of representation error is usually referred to as "roundoff error". Most numbers cannot be represented exactly and must be perturbed slightly to numbers that are representable. In some sense truncation error is misrepresentation of a mathematical operation while representation error refers to misrepresentation of the data.

Certainly we want to deal with the ultimate source of each error in a way that minimizes the error as it enters the problem. But of much greater consequence is the factor by which errors are magnified after they occur. A classical example of error magnification occurs in Wilkinson's polynomial. This polynomial is

$$p(z) = (z-1)(z-2)\cdots(z-20) = z^{20} - 210z^{19} + \ldots + 20!$$

and some of the roots of this polynomial (*i.e.*, $1, 2, 3, \ldots, 19, 20$) are very sensitive to perturbations in values of the coefficients of the polynomial. For example, if you change the coefficient of z^{19} from -210 to $-210 + 10^{-7}$, then the roots change dramatically, as shown in Figure 1.

Before we can discuss error magnification conveniently we need to set up a theoretical framework for it.

■ 1.1 Precision and Accuracy

The terms *precision* and *accuracy* in *Mathematica* have a technical meaning. Roughly speaking, precision refers to the number of significant digits in a number and accuracy refers to the number of significant digits to the right of the decimal point. Since the least significant digit of a number can be to the *left* of the decimal point it is possible for a number to have negative accuracy. In fact there are three cases and we give examples of each.

Define a function that gives both the precision and the accuracy of a number.

```
In[1]:= f[x_] := {Precision[x], Accuracy[x]}
```

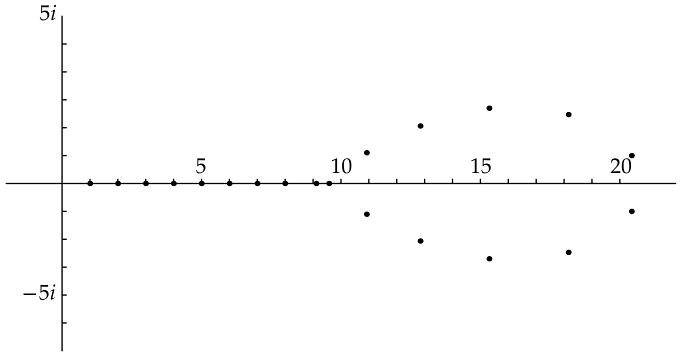

Figure 1: Effect of a perturbation on roots of a polynomial

This number has 27 significant digits, 16 of which are to the right of the decimal point.

```
In[2]:= f[33333333333.3333333333333333]

Out[2]= {27, 16}
```

This number has 20 significant digits and the least significant digit is in the 31st decimal place.

```
In[3]:= f[0.000000000003333333333333333333]

Out[3]= {20, 31}
```

This number has 19 significant digits and the least significant digit is is in the -7th decimal place. *I.e.* there are 7 insignificant digits beyond the least significant digit and before the decimal point.

```
In[4]:= f[3.333333333333333333 10^25]

Out[4]= {19, -7}
```

It turns out that although counting significant digits appears quite simple and intuitive it has definite problems from a mathematical point of view, the most important being its discontinuities. To avoid such problems precision and accuracy are actually defined in terms of error bounds. A floating-point number in *Mathematica* can be thought of as having some error associated with it. If we knew what this error was we could correct for it and get back the true value. In general this is not possible and all that we know about it is some error bound. For example, in the uncertain number $123.47 \pm .003$ the error bound is .003. But the error bound can be used to give mathematically sound definitions of precision and accuracy:

accuracy of $x \pm \epsilon$	$-\log_{10} \lvert\epsilon\rvert$
precision of $x \pm \epsilon$	$-\log_{10} \lvert\epsilon/x\rvert$

Definitions of precision and accuracy.

In particular, accuracy refers to the absolute error bound and precision refers to the relative error bound.

Clearly precision and accuracy do not need to be integers. In fact making them integers is much like trying to count significant digits. But for reasons of consistency with older versions, when you use the functions `Precision[ ]` and `Accuracy[ ]` you get integer results: the real value gets rounded to the nearest integer before being returned. This may change in future versions[1] of *Mathematica*.

In our discussion below we will find it convenient to define one more term: *magnitude*. The magnitude of a number is the base 10 logarithm of its absolute value.

[1]You can turn off rounding in versions 2.1 and later by entering `SetPrecision[Round, False]`. Although semantically this makes no sense, this particular pair of arguments to `SetPrecision[ ]` causes rounding to be turned off.

1.2 Condition Numbers

In an effort to quantify the magnification of errors we consider first a function of a single variable. If $y = f(x)$ where x and y are the "correct" values and Δx and Δy are their respective errors, then

$$\Delta y \approx f'(x)\ \Delta x$$

The number $f'(x)$ is the *absolute condition number* and is the factor by which an error in x gets magnified to become an error in y. In terms of accuracy we have that the accuracy of $f(x)$ is the accuracy of x less the magnitude of the absolute condition number of f at x.

Since fixed-point numbers have been replaced almost universally by floating-point numbers, the absolute condition number of a function is not particularly important. A much more important concept is the *relative condition number* or simply the *condition number* of a function. The condition number is the factor by which the relative error in x gets magnified to become a relative error in y. Since

$$\frac{\Delta y}{y} \approx \frac{x\,f'(x)}{f(x)}\,\frac{\Delta x}{x}$$

we have $x\,f'(x)/f(x)$ as the condition number. In terms of precision we have that the precision of $f(x)$ is the precision of x less the magnitude of the condition number of f at x.

For a function f that maps several variables to several variables, the Jacobian

$$\left[\frac{\partial \mathrm{f}}{\partial \mathrm{x}}\right]$$

of the function is the *absolute condition matrix* and the relative condition matrix is similar with elements multiplied by x_j/y_i. The condition matrix is rarely if ever used in practice. In numerical linear algebra it is often convenient to simplify matters by using norms to replace the condition matrix by a single condition number that bounds the error magnification of the norms of vectors and matrices.

This model is still not really complete. In particular it is only a first order approximation to what is really going on and this is not adequate to analyze all problems. For example, consider the values of the roots of the polynomial $x^2 - 2x + 1 - \varepsilon$. The roots are $1 + \sqrt{\varepsilon}$ and $1 - \sqrt{\varepsilon}$ and the condition number (with respect to $1 - \varepsilon$) of computing them from the coefficients of the polynomial is about $1/(2\sqrt{\varepsilon})$. If ε is 0 the sensitivity of the roots to perturbations in ε is infinite, but that does not imply that a finite error in ε results in an infinite error in the values of the roots.

There is a common misconception that subtraction of two very nearly equal numbers is bad because it introduces a large "roundoff" error. In fact subtraction of two nearly equal numbers is one of the few places where there is absolutely no new error introduced. The problem with subtraction is that when the operands are nearly equal the condition numbers are very large: if $z = x - y$, the condition number with respect to x is x/z and the condition number with respect to y is $-y/z$. The difference $x - y$ is very sensitive to all *previous* errors (*i.e.*, data error) in x and y. In the unlikely event that there are no previous errors, the result has no error either. It is, however, quite likely that the data error in any number is at least half of the value represented by the least significant bit of that number.

Although the above framework has its limitations, it is nevertheless useful in that it explains to a large extent how errors propagate. You should of course realize that each operation not only propagates all data error, but also introduces its own representation and truncation errors.

1.3 Condition of an Algorithm

The condition or numerical stability of an algorithm is a more qualitative description of an algorithm. An algorithm can be well-conditioned or ill-conditioned, but it is difficult to assign a number to the condition. This is because the error in the result is more than a magnification of the data error; new errors get introduced as the algorithm proceeds. Of course there is a condition number corresponding to each of those individual errors that measures the magnification of that error in the final result, but an analysis of an algorithm sufficiently detailed to reveal these condition numbers is rarely possible or even practically useful.

It is important to recognize that while there may be several algorithms to compute a given function they may be conditioned very differently. This idea is best explained with an example. Consider the function $y = e^x$ and two ways of evaluating it:

$$y_1 = \sum_{i=0}^{\infty} \frac{x^i}{i!}$$

and

$$y_2 = \left(\sum_{i=0}^{\infty} \frac{(-x)^i}{i!} \right)^{-1}$$

If we wish to evaluate y at $x = -50.0$ we get, in either algorithm, terms as large as 2.92×10^{20}. In y_2 all of the terms are positive and the sum is about 5.2×10^{21}, the final result being about 1.93×10^{-22}. Notice, however, that in y_1 the terms alternate in sign and the smallness of the result is achieved by *cancellation* of the leading 42 digits in the largest terms; if you use less than 42 digits of precision the result cannot possibly be relied upon to have even the first digit correct. On the other hand, the result y_2 can be expected to have very little loss of precision. Of course for $x = +50.0$ the conditioning of the two algorithms is reversed.

Now both y_1 and y_2 involve infinite sums. In practice there must be truncation error incurred in making either algorithm finite. Notice that for $x = -50.0$, y_1 requires nearly 180 terms just to ensure that the individual terms are smaller than the sum. In y_2, the use of 180 terms is sufficient to achieve 45 digits of precision (assuming the arithmetic is sufficiently precise).

The point of all this is that it is important to consider the condition of an algorithm. Certainly it is possible to use very high-precision arithmetic and algorithms with very small truncation error (*e.g.* many, many terms of an infinite sum), but an ill-conditioned algorithm may take thousands of times as long as a well-conditioned algorithm to achieve a result with comparable error. Of course if the function itself is ill-conditioned then no amount of effort in choosing a good algorithm can result in a well-conditioned algorithm; the condition number of a function is a lower bound for the condition of an algorithm.

Finally, even if you could find the perfect algorithm with which to evaluate e^x (*i.e.*, an algorithm that introduced no errors of its own, but only propagated the data error) and you asked it to evaluate e^1 (no data error), it would still have to represent the final result. Since e cannot be represented exactly there would still be representation error.

2 Arithmetic

At this point we digress somewhat to explain the arithmetic used in *Mathematica*.

■ 2.1 Introduction

You may wonder what there could possibly be to explain about arithmetic. After all, everyone understands what calculators do.

For exact arithmetic, *i.e.*, integer and rational arithmetic, there is indeed little that needs to be said. Once a canonical form for the numbers is chosen, there is only one right answer. However, for inexact arithmetic there are several important points, some of which are rather subtle, that you need to be aware of. First, unlike a calculator, *Mathematica* can handle numbers of any precision—not just the 10 digits or so that fit in the calculator display. Second, *Mathematica* deals with a much larger and more complicated set of mathematical functions. Finally, *Mathematica* uses two fundamentally different types of inexact arithmetic.

Just to show why one has to be careful about arithmetic, let's start with a fairly horrifying example. The problem is to calculate the value of $\sin(10^{30})$. Try it on any calculator—you'll get the wrong answer. Likewise, just typing `N[Sin[10^30]]` in *Mathematica* will give you a silly answer. But, there is a difference! In

Mathematica the `Sin[ ]` function does the best that it can. In fact it gives the correct result for whatever argument it receives. The problem is that it requires 70 bits to correctly represent the mantissa of 10^{30}. When you ask for `N[10^30]` you implicitly ask *Mathematica* to use machine-precision numbers, which on many machines have only 53 bits in the mantissa.

Let `x` be the machine number that is closest to the number 10^{30}.

```
In[5]:= x = N[10^30]

              30
Out[5]= 1. 10
```

Let `y` be the same number as `x`, but with 60 digits of precision.

```
In[6]:= y = SetPrecision[x, 60]

                                                      30
Out[6]= 1.00000000000000000019884624838656 10
```

Note that our number is quite far from 10^{30}.

```
In[7]:= y - 10^30

                               13
Out[7]= 1.9884624838656 10
```

The value of `Sin[x]` is correct, but it is different from $\sin(10^{30})$ because x is quite different from 10^{30}.

```
In[8]:= {Sin[x], Sin[y], N[Sin[10^30], 60]}

Out[8]= {0.00933147, 0.00933146893117582405723721943,
  -0.0901169019121380580303864289 53}
```

These results can be very confusing unless one understands the ways *Mathematica* does arithmetic. The next section explains some of the important points of arithmetic that you need to be aware of to use the high-precision arithmetic in *Mathematica* effectively.

2.2 Types of Inexact Arithmetic

There are several ways to do arithmetic when the quantities cannot be represented exactly. In the past, people sometimes used fixed-point arithmetic because it was thought that the analysis of the resulting errors was simpler. This assumption has been shown to be false, and fixed-point arithmetic is now nearly nonexistent except in special purpose applications. Another way to do arithmetic is to use interval arithmetic. In this system inexact numbers are represented by intervals; the number being represented lies somewhere within the interval and the shorter the interval the more precisely the number is known. For complex numbers the interval generally becomes either the Cartesian product of the intervals for the real and imaginary parts or a disc specified by a center and a radius. *Mathematica* uses neither fixed-point arithmetic nor interval arithmetic for inexact calculations[2].

The most common way to do inexact arithmetic is to use fixed-precision floating-point arithmetic. In this system a number is stored as a mantissa and an exponent. The mantissa always contains the same number of bits and, since there is no way to tell how precise a number is, all numbers are treated as exact. The problem with this is that the arithmetic is necessarily wrong. For example, if we are working within a system where all numbers have five significant digits, then $1.2345 + .012345$ cannot be correctly represented. In order to represent the answer, it must be rounded to five significant digits; the effect of this rounding is to make the arithmetic wrong. Another instance of this problem is exhibited in the following *Mathematica* example:

Define a function.

```
In[1]:= f[x_] := 1/(x-1)
```

Notice that there is some error. (With exact arithmetic the result would be 10^{13}.)

```
In[2]:= f[1+10.^-13]

                  13
Out[2]= 1.0008 10
```

The error is in fact quite large and could be made larger by evaluating `f[ ]` at a number even closer to 1.

```
In[3]:= %-10^13

                   9
Out[3]= 7.99917 10
```

[2]We are talking here about the standard arithmetics built into *Mathematica*. The package `NumericalMath`InteralAnalysis`` does provide rigorous, outwardly rounded interval arithmetic.

The primary way that errors get introduced in this arithmetic is by trying to represent numbers that have no representation in the system. For example, the number $\frac{1}{3}$ cannot be represented exactly as a floating-point number[3]. The advantage of fixed-precision floating-point arithmetic is that it is relatively easy to implement in hardware (making it very fast) and the behavior of the errors thus introduced is well understood. Because this type of arithmetic is often done in hardware we refer to it as machine arithmetic and the numbers as machine numbers. *Mathematica* uses this type of arithmetic when it does machine-precision arithmetic.

An alternative to fixed-precision floating-point arithmetic is *variable*-precision floating-point arithmetic or *significance arithmetic*. In this system the numbers are thought of as being fuzzy (much like in interval arithmetic), but the arithmetic on these fuzzy numbers is correct in the sense that no errors are introduced by the arithmetic itself. For example, if x is the fuzzy number 1.2345?????... (the trailing "?"s indicate that these digits are unknown) and y is the fuzzy number .12?????..., then $x + y$ is the fuzzy number 1.35?????.... Furthermore, we don't have to lie about a number like $\frac{1}{3}$. Any number can be represented to any precision; whatever the precision of the representation, the amount of uncertainty in the number is implicit in that representation. Continuing with the example above, in variable-precision arithmetic we see that the answer is correct, including the precision:

No error is exhibited.

```
In[4]:= f[1+N[10^-13,20]]

             13
Out[4]= 1. 10
```

The precision is just what we expect.

```
In[5]:= Precision[%]

Out[5]= 20
```

The disadvantages of this type of arithmetic are that is must be done in software, analysis of the error propagation is not generally as well understood, and the error can be overly pessimistic. For example, if x is again the fuzzy number 1.2345?????..., then $x - x$ is the fuzzy number .0000?????.... Clearly, whatever number is represented by the fuzzy number x, $x - x$ should be *exactly* 0. Because this type of arithmetic deals with arbitrarily high-precision numbers, we often refer to it as big number arithmetic and the numbers as bignums. *Mathematica* uses this type of arithmetic when it does high-precision arithmetic.

We are now in a position to explain the example in the introduction to this section. To compute the sine of a large number, some multiple of 2π must first be subtracted to reduce the number to the interval $[0, 2\pi)$. For machine-precision numbers this is exactly what is done. It does not matter that the least significant bit in the mantissa corresponds to billions of periods of the sine function; the number is still well defined even though it is extremely unlikely that it represents a number in which you are interested. Once the argument is reduced and converted back to a machine number the sine is evaluated and the result is a machine number, although the result is not necessarily correct to the last bit. For big numbers the situation is quite different. Here $159154943091895335768883763372 \times 2\pi$ is subtracted from 10^{30} leaving $3.23183197748\ldots$. Using 60 digits of precision, we get an answer with 29 digits of precision; 31 digits being lost in the process of subtracting the huge multiple of 2π from 10^{30}. Another way of seeing this is to consider the condition number of $\sin x$ at $x = 10^{30}$: $10^{30}\cot(10^{30}) = 1.1\ldots10^{31}$. The relative error in the argument will be magnified about 10^{31} in the result and 31 digits will be lost. Of course, if you start with less than 32 digits of precision in x, the result will have no digits at all.

It remains to explain the interaction of the two types of arithmetic. Arithmetic with numbers of either type results in a number of the same type with the exception of overflow and underflow in machine-precision arithmetic: if a number cannot be represented in the format for machine numbers, the operands are converted to bignums and the operation is performed using big number arithmetic. Since machine numbers usually contain errors of unknown magnitude and since machine arithmetic is much faster than big number arithmetic, when an operation is performed on a mixture of machine numbers and bignums, the bignums are first coerced to machine numbers and the operation is performed using machine arithmetic. If this is not possible, big number arithmetic is used. It should be noted here that even if the precision of a bignum is less than machine precision, the number remains a bignum; to coerce it to a machine number would require that the information on the magnitude of the fuzz be discarded.

[3]We are concerned here with binary and decimal number systems. Clearly this is a false statement in a ternary number system.

2.3 More Details of Big Number Arithmetic

The above discussion is not the complete story on big number arithmetic. One problem with such arithmetic is that the precision of a sum can be very big.

A simple arithmetic problem can result in a very large answer.

```
In[1]:= 1 + 10.^-500

Out[1]= 1.00000000000000000000000000000000000000000000000000000\
  0000000000000000000000000000000000000000000000000000000000000\
  0000000000000000000000000000000000000000000000000000000000000\
  0000000000000000000000000000000000000000000000000000000000000\
  0000000000000000000000000000000000000000000000000000000000000\
  0000000000000000000000000000000000000000000000000000000000000\
  0000000000000000000000000000000000000000000000000000000000000\
  0000000000000000000000000000000000000000000000000000000000000\
  00000000000000000001
```

In fact if something had not been done to deal with this problem it would be very easy to cause *Mathematica* to crash due to insufficient memory on very simple problems. To avoid such problems the global variable `$MaxPrecision` is used to place an upper bound on the precision of the numbers that *Mathematica* uses. The default value for `$MaxPrecision` is `50000`.

If it weren't for `$MaxPrecision` this would be more than 100000.

```
In[2]:= Precision[1 + 10.^-100000]

Out[2]= 50000
```

You can rather dramatically change the behavior of the big number arithmetic in *Mathematica*. In addition to `$MaxPrecision` there is global variable for the lower bound of the precision: `$MinPrecision`. `$MinPrecision` and `$MaxPrecision` bound the precision of all results. In fact big number arithmetic is *not* arbitrarily high-precision arithmetic. Unless you change the value of `$MaxPrecision` no (inexact) result will ever have more precision than 50000. By changing the values of `$MinPrecision` and `$MaxPrecision` to be the same, say 100 you can get fixed, high-precision arithmetic. You should be aware that there is a difference between this arithmetic and machine arithmetic (other than the precision). All bignum arithmetic in *Mathematica* uses hidden digits, *i.e.*, digits that are included in the calculations but are neither displayed nor considered significant. When loss of digits occurs due to cancelation of leading digits a positive value of `$MinPrecision` may cause some of the hidden digits to become exposed as the least significant digits of the result. If there are already at least `$MinPrecision` digits in the number no hidden digits will be exposed. With machine arithmetic there are no hidden bits and bits lost due to cancelation are made up for by making the missing least significant bits to be 0s.

Start with two slightly different numbers.

```
In[1]:= a = N[1/3, 40]; b = N[1/3 - 10^-35, 40];
```

Make the arithmetic be 30 digits, fixed-precision.

```
In[2]:= $MinPrecision = $MaxPrecision = 30;
```

The values of `a` and `b` appear to be the same.

```
In[3]:= {a, b}

Out[3]= {0.333333333333333333333333333333,
  0.333333333333333333333333333333}
```

If you subtract them the hidden digits get uncovered.

```
In[4]:= a - b

                                                -35
Out[4]= 1.000000000065889137128973506261 10
```

It is not recommended that you use fixed-precision, bignum arithmetic unless you want to do your own error analysis. Setting `$MinPrecision` to a positive value will force all results to be at least that precise even if such precision cannot be justified. Setting `$MaxPrecision` to a value lower than its default may be useful in some cases and it will not result in digits that are not justified.

■ 2.4 SetPrecision[ ] and SetAccuracy[ ]

`SetPrecision[ ]` and `SetAccuracy[ ]` are used to control the magnitude of the fuzz in a number. Note that these functions do not affect the actual error in a number; such behavior is not practical. Nor do they change their arguments in any way; they *return* a value that is a modified version of their first argument. With expressions `SetPrecision[ ]` and `SetAccuracy[ ]` return results in which all numbers have been converted to big numbers.

The description above of the way fuzz is represented is an oversimplification. The unknown digits in a number are not in fact "?"s or any other nondigit. Bignums are represented in binary rather than decimal with finitely many explicit bits followed by infinitely many implicit zeros. There is also a number associated with each bignum that indicates where the unknown bits are supposed to begin. The known bits may include implicit zeros or the unknown bits may include explicit bits. Internally the precision of a number is measured in bits, although as a user you only see precision measured in decimals. When `SetPrecision[ ]` is applied to an exact number (*i.e.*, an integer or a rational number), a bignum is created with a few more bits than are requested and the unknown bits are so indicated. These explicit, hidden bits are in fact used in the subsequent calculations, but the boundary between the known and unknown digits is calculated for each result.

When `SetPrecision[ ]` is applied to an inexact number, the only thing that happens is that the boundary between the known and unknown digits is moved to get a result with the requested precision. This is easily demonstrated as follows:

Start with a bignum and 40 digits of precision.

```
In[1]:= a = SetPrecision[1/3,40]

Out[1]= 0.3333333333333333333333333333333333333333
```

More than 40 digits were actually correct.

```
In[2]:= b = SetPrecision[a,60]

Out[2]= 0.33333333333333333333333333333333333333333333333310525\
  7411405
```

Lower the precision to 20 digits.

```
In[3]:= c = SetPrecision[a,20]

Out[3]= 0.33333333333333333333
```

Now raise the precision to 60 digits. Note that none of the digits was destroyed when `c` was formed.

```
In[4]:= d = SetPrecision[c,60]

Out[4]= 0.33333333333333333333333333333333333333333333333310525\
  7411405

In[5]:= e = SetPrecision[1/3,20]

Out[5]= 0.33333333333333333333
```

Raising the precision of `e` to 40 before adding `d` causes 40 of the digits of `d` to be used in the sum.

```
In[6]:= SetPrecision[e,40] + d

Out[6]= 0.6666666666666666666666666666624594085055
```

Forming the sum first uses only those digits of `d` which correspond to known digits in `e`.

```
In[7]:= SetPrecision[e+d,40]

Out[7]= 0.6666666666666666666666666666624593443078
```

It should be noted that the implicit, trailing zeros in a bignum are binary zeros rather than decimal zeros. The effect of this is that when you increase the precision of a number you often expose (decimal) digits that are not zeros. The simplest way to explain this is to consider the number 1/10. In decimal this number is 0.1000 . . . . In binary it is the periodic expansion 0.0001100110011 . . . . Clearly, if you take a finite precision approximation to this number and extend its precision with binary zeros you will end up with a number that is different from 0.1000 . . ., and nonzero decimal digits will appear in the result.

■ 2.5 N[ ]

The functions `SetPrecision[ ]` and `SetAccuracy[ ]` only work on numbers, although they do recursively go inside of complicated expressions and do their thing on all numbers that they encounter. But some expressions are not numbers even though they may represent numerical quantities. For example `Pi` is not a number in *Mathematica*; it is simply a symbol. If you enter `SetPrecision[Pi, 20]` you just get `Pi` back. To get a numerical approximation to π you need to use the function `N[ ]`. `SetPrecision[ ]` and `SetAccuracy[ ]` are very simple in their operation: any numbers they encounter in an expression are simply converted to bignums. Of course once numbers become bignums the evaluator often takes over and makes further changes until nothing more can be done, but the operation of `SetPrecision[ ]` and `SetAccuracy[ ]` is quite simple. The function `N[ ]` on the other hand is more complicated. There are internal rules for what `N[ ]` should do with certain symbols like `Pi` and `Integrate`. In addition you can add your own rules to symbols describing what `N[ ]` should do with them.

Another major difference between `N[ ]` and `SetPrecision[ ]` is that `N[ ]` will never raise the precision of anything and may give a result with less precision than was requested. `SetPrecision[ ]` gives whatever precision is specified, even if it has to use digits that are incorrect.

Define `a` to be the difference of two nearly equal quantities.

```
In[1]:= a = Sin[1] - 295031332/350613791

          295031332
Out[1]= -(---------) + Sin[1]
          350613791
```

The argument of `Sin[ ]` becomes a bignum and then `Sin[ ]` evaluates to a number. Since the difference is small some 18 digits are lost due to cancellation.

```
In[2]:= N[a, 30]

                           -18
Out[2]= 2.661305922855 10
```

Here again digits are lost to cancellation, but the precision of the result is set to 30 before returning.

```
In[3]:= SetPrecision[a, 30]

                                              -18
Out[3]= 2.661305922855443049931748738503 10
```

Here we see what the correct result to 32 digits is (again some 18 digits were lost, but we started with 50 this time.) We see that about 10 of the extra digits in the previous result were in fact correct, but the last 8 were wrong.

```
In[4]:= N[a, 50]

                                                -18
Out[4]= 2.66130592285544304993175355125458 10
```

The function `N[ ]` can be used with a single argument. If you don't specify the precision to use or you specify a value that is less than the precision of machine numbers (*i.e.*, `$MachinePrecision`) `N[ ]` will attempt to return machine numbers when it encounters numerical quantities. There is a field in the data structure for machine numbers that specifies how many digits to print. This is quite different from the precision of a number. The precision of any nonzero machine number is just `$MachinePrecision` and is fixed by the manufacturer of the computer. The number of digits to be printed for a machine number is determined by the number of digits entered when the number was typed in or the second argnument to `N[ ]` if the number was created by `N[ ]`. If the number was created by any arithmetic operation it doesn't know how many digits to print and so by default prints six. In any case trailing zeros, even if they are significant, are not included.

Form a table of machine numbers displayed with various numbers of digits.

```
In[1]:= a = Table[N[Pi, k], {k, 5}]

Out[1]= {3., 3.1, 3.14, 3.142, 3.1416}
```

Each of those machine numbers is the same value. The only difference is the number of digits to be printed.

```
In[2]:= a - N[Pi]

Out[2]= {0., 0., 0., 0., 0.}
```

Even the simplest arithmetic operation causes the information on the number of digits to be printed to be lost. In such cases 6 digits are displayed.

```
In[3]:= -a

Out[3]= {-3.14159, -3.14159, -3.14159, -3.14159, -3.14159}
```

The expressions `N[0]` and `N[0, prec]` present special problems. We know that `N[0]` should not be the machine number `0.0` because the precision of `0.0` is too small; it is in fact 0. In fact *any* inexact zero has a precision of 0 and even an *exact* zero has only an indeterminate precision because its relative error is indeterminate. So from a purist point of view making `N[0]` give `0` is at least the least incorrect thing to do. In fact that is what is done by default. At the same time we must recognize that users would often like `N[0]` and `N[0, prec]` to do something different, although just what that might be is unclear. To deal with this problem of ambiguity there is the global variable `$NZero` which can be set to a function for evaluating `N[0]` and `N[0, prec]`

Define a function to evaluate `N[0]` and `N[0, prec]`. Notice the use of a default value for the second arg. We need to make it accept both the single argument `0` and `0` with a precision value.

```
In[4]:= Default[foo, 2] = 0; foo[0, prec_.] :=
          If[prec <= $MachinePrecision, 0.0,
          SetAccuracy[0, prec+20]]
```

Set the value of `$NZero` to be `foo`.

```
In[5]:= $NZero = foo

Out[5]= foo
```

Try evaluating `N[ ]` at `0` in several different ways.

```
In[6]:= {N[0], N[0, 10], N[0, 30]}

Out[6]= {0, 0, 0}
```

Only the last value is a bignum.

```
In[7]:= MachineNumberQ /@ %

Out[7]= {False, False, False}
```

Finally, there is a means of protecting expressions from the effects of `N[ ]`. Just as `SetPrecision[ ]` and `SetAccuracy[ ]` search through an entire expression, converting all numbers to bignums with the requested precision or accuracy, `N[ ]` searches through an entire expression using any rules that apply. The details of how this is done is explained in The *Mathematica* Book. But you can also prevent `N[ ]` from going in and doing its thing where it is not wanted. This is done by setting the attributes of a symbol to include one of `NProtectedAll`, `NProtectedFirst`, or `NProtectedRest`. This protection feature is rarely needed, but it is quite useful when it is needed.

Make `f[ ]` not allow `N[ ]` to do anything to its first argument.

```
In[1]:= SetAttributes[f, NProtectedFirst]
```

`N[ ]` does not get to touch the first argument of `f[ ]`.

```
In[2]:= N[f[Pi + 10/3, E - 2], 20]

          10
Out[2]= f[-- + Pi, 0.7182818284590452354]
          3
```

The `N[ ]` protection attributes have no effect on `SetPrecision[ ]` and `SetAccuracy[ ]`.

```
In[3]:= SetPrecision[%, 10]

Out[3]= f[3.333333333 + Pi, 0.7182818285]
```

■ 2.6 The Problem of All Errors Being Treated as Independent

We mentioned earlier that if you take a bignum x and subtract it from itself you do not get an exact 0, but rather a zero whose accuracy is comparable to the accuracy of x. We examine here a more complicated example and describe one way to combat the problem. Consider the following attempt to evaluate `ChebyshevT[200, 5/7]` using the three-term recurrence relation for `ChebyshevT[ ]`:

Define the function.

```
In[4]:= chebyT[n_,z_] := Module[{t0=1, t1=z, tmp},
                              Do[ tmp = 2 z t1 - t0;
                                     t0 = t1;
                                     t1 = tmp,
                                     {i, 2, n}
                                     ];
                              t1]
```

Evaluate it at N[5/7, 60]

```
In[5]:= y = chebyT[200, N[5/7, 60]]

Out[5]= -0.45

In[6]:= Precision[y]

Out[6]= 2
```

Clearly, the answer has much lower precision than the 60 digits with which the argument z was given. Such loss of precision in itself is not necessarily overly pessimistic; perhaps the (relative) condition number of the function is very large. However, in this case the condition number is in fact only slightly more than 400; only two or three digits should be lost in a good algorithm. The problem is that, while there is really only one degree of freedom associated with the fuzz in z, *Mathematica* treats the fuzz in t1 and t0 for each value of i, as independent. Then, applying the triangle inequality with each iteration, the fuzz in the final answer is quite large. It should be noted here that machine-precision arithmetic, which simply ignores its representation errors, actually gets a much better result than that of variable-precision arithmetic.

The error is very small.

```
In[7]:= chebyT[200, N[5/7]] - ChebyshevT[200, N[5/7, 100]]

                         -15
Out[7]= -5.32907 10
```

Book way to deal with this loss of precision is just to start with more precision than is desired in the final result. We did this just now to find a good value for comparison of the machine-precision result. For many problems, the number of digits lost is independent of the precision used and the number of digits lost is not nearly so many as in the above example. This approach is often acceptable for small problems, but for larger problems it can be very expensive.

Another way to deal with loss of precision is to force variable-precision arithmetic to behave more like fixed-precision floating-point arithmetic. We can do this by setting $MinPrecision and $MaxPrecision. This approach is exhibited below.

Fix the precision of bignum arithmetic.

```
In[8]:= $MinPrecision = $MaxPrecision = 50

Out[8]= 50
```

Evaluate it at N[5/7, 50]]

```
In[9]:= chebyT[200, N[5/7, 50]]

Out[9]= -0.45303043768986593492854596910003821853780284803122
```

Of course, in the process of forcing the precision to be higher than that resulting from variable-precision arithmetic, we have destroyed its ability to estimate the precision of the result for us. We must now go back and analyze how each of the errors incurred along the way contributes to the error in the final result. It turns out that in this example the condition number of the function itself provides a reasonably good estimate of magnification of each of the errors except for those values of the argument where the condition number is close to zero. In particular the final result in the last example should have its precision reduced to about 47.

One problem with setting $MinPrecision and $MaxPrecision to values other than their default values is that you may forget that you have done so and not realize that the results you are getting contain unjustified digits. A better way to deal with fixed precision is to use it only inside a Block[ ]. When *Mathematica* exits from the Block[ ] the old values of $MinPrecision and $MaxPrecision are restored.

Define the function.

```
In[1]:= chebyT[n_,z_] := Block[{$MinPrecision = 50,
                                 $MaxPrecision = 50},
                            Module[{t0=1, t1=z, tmp},
                               Do[ tmp = 2 z t1 - t0;
                                  t0 = t1;
                                  t1 = tmp,
                                  {i, 2, n}];
                               t1]]
```

This result is the same as what we got by fixing the precision.

```
In[2]:= y = chebyT[200, N[5/7, 50]]

Out[2]= -0.45303043768986593492854596910003821853780284803122
```

`$MinPrecision` and `$MaxPrecision` still have their default values.

```
In[3]:= {$MinPrecision, $MaxPrecision}

Out[3]= {0., 50000.}
```

In *Mathematica* all of the elementary transcendental functions as well as all of the orthogonal polynomials[4] give exactly those digits that are justified by the precision of the argument, except that the precision given by an orthogonal polynomial will not exceed the precision of the argument. Of course this only applies when the functions are evaluated using variable-precision arithmetic; with fixed-precision arithmetic you get a result with precision determined by `$MinPrecision` and `$MaxPrecision`.

3 Heuristics for Controlling Numerical Errors

We cannot possibly examine all algorithms and choose the best one for a particular application; there are too many possibilities. We present here some heuristics that are useful in choosing and designing algorithms.

■ 3.1 Large Sums

For large sums with numbers varying greatly in magnitude the traditional wisdom is to sum the terms starting with those that are smallest in magnitude and ending with those that are largest in magnitude. This is good advice for fixed-precision arithmetic. For the variable-precision arithmetic used by *Mathematica* the ordering of the terms is essentially irrelevant since the error in the sum is the sum of the errors in each of the terms. Moreover, for fixed-precision arithmetic, if there are terms of opposite sign then the preferred arrangement is to pair up terms that will tend to cancel each other. The reason for this is that, as we discussed at the end of Section 1.2, subtractive cancellation is rather ill-conditioned. The earlier in the calculation it occurs the less data error there is likely to be and the less error in the result. The reason deferring subtractive cancellation with variable-precision arithmetic in a large sum is not a problem is that variable-precision arithmetic does not make representation errors (unless it runs into `$MaxPrecision` or `$MinPrecision`) that will be magnified later. Neither of these two suggestions is likely to result in more than a few more correct decimals unless the sums contain an extremely large number of terms. In fact, in rare cases the preferred order of summation can give a result with *more* error than some other order.

It should be noted that `Plus[ ]` is `Orderless` so its arguments are first sorted in order of smallest to largest. It needs to do this to have them in a canonical order so that like terms can be combined in those cases where the result will not be a simple number. If the terms are all positive and have the same precision then this is also the best order to sum them numerically. Of course if they are all negative and have the same precision then this is exactly the opposite of the best order in which to sum them. Also, `Sum[ ]` works by first forming a list of the terms and then applying `Plus[ ]` to the list. So for really critical sums where you need control over the order in which the terms are included, you should use `NSum[ ]` with the option `NSumTerms` set at least as large as the total number of terms. (This prevents `NSum[ ]` from trying to use fancy techniques to evaluate the sum.) But `NSum[ ]` is designed primarily for infinite sums where the terms tend to *decrease* in magnitude. Thus it sums the first `NSumTerms` terms in reverse order and you need to arrange that the terms be put in opposite order from the order in which you want them included.

Sum the terms from smallest to largest.

```
In[1]:= a = NSum[1/j, {j, 10000}, NSumTerms -> 11000]

Out[1]= 9.78761
```

[4] We refer here to orthogonal polynomials that are actually polynomials and not associated functions, *e.g.*, `ChebyshevT[23/7,z]` is not included. Furthermore, evaluating `ChebyshevT[27,z]` symbolically and then replacing `z` by some inexact number may result in a number of much lower precision than evaluating `ChebyshevT` at that number directly. This is because the polynomial expression represents a very ill-conditioned algorithm.

Sum the terms from largest to smallest.

```
In[2]:= b = NSum[1/j, {j, 10000, 1, -1}, NSumTerms -> 11000]

Out[2]= 9.78761
```

The first sum has less error.

```
In[3]:= {a, b} - NSum[1/j, {j, 10000}, NSumTerms -> 11000,
                    WorkingPrecision -> 25]

                    -15            -14
Out[3]= {3.55271 10    , -3.73035 10   }
```

3.2 Large Condition Numbers First

The suggestion that sums be arranged so that cancellation of digits occur as early as possible is a special case of a more general rule. The rule is that if you have a choice between two algorithms both of which have an ill-conditioned part, use the algorithm in which the ill-conditioned part occurs as early as possible. The reason for this is that as an algorithm proceeds, the data tend to include increasingly more error. If some part of the algorithm has a large condition number, all of the error in its input data will be magnified many times. The error that occurs *after* the large magnification is not nearly as serious.

The ill-conditioned part (exponentiation) occurs last.

```
In[1]:= a = 2^N[1000 + E]

                   301
Out[1]= 7.05148 10
```

The ill-conditioned part here is first.

```
In[2]:= b = 2^N[1000] 2^N[E]

                   301
Out[2]= 7.05148 10
```

The relative error in a is larger.

```
In[3]:= {a, b}/2^N[1000 + E, 25] - 1

                    -14
Out[3]= {-3.23075 10   , 0.}
```

3.3 Appropriate Data Structures

It is important to design the data structures for your problems so that the data they contain will have a minimum of redundancy. If there are five parameters in a problem and they interact in such a way that two of them always contain nearly the same information, then essentially you are wasting one of the parameters. In some abstract sense, the data should be as orthogonal as possible. This is best seen with some examples. The first example is rather contrived, but be patient; the second example is much more natural.

A Trivial Example

Consider the linear vector space $\mathbf{R}^2$. If we use as a basis for this space the pair of vectors $\{b_1 = (1346269, 2178309), b_2 = (2178309, 3524578)\}$ we are likely to lose accuracy when we use inexact linear combinations of these numbers.

Define the basis.

```
In[1]:= basis = {{1346269,2178309},{2178309,3524578}};
```

Find the exact coefficients required to represent the point (π, e).

```
In[2]:= LinearSolve[basis, {Pi, E}]

          -(2178309 (1346269 E - 2178309 Pi) - Pi)
Out[2]= {-----------------------------------------,
                          1346269

  1346269 E - 2178309 Pi}
```

This gives machine-precision values for the coefficients. We calculate them using 40 digits of precision to ensure that the problem is not in the evaluation of the coefficients themselves; the coefficients are correct to the last decimal place.

```
In[3]:= coef = N[N[%, 40]]

                   6             6
Out[3]= {5.15153 10 , -3.18382 10 }
```

These numbers don't look quite right.

```
In[4]:= basis . coef

Out[4]= {3.1416, 2.71875}
```

Indeed, there is a large error.

```
In[5]:= % - N[{Pi, E}]

                     -6
Out[5]= {8.90891 10   , 0.000468172}
```

Another Example

The above example was simple enough that it was fairly obvious just where the problem was in terms of the elementary arithmetic operations used. In a more abstract sense the problem arises from the fact that the basis for the space is not orthogonal, or more intuitively, the basis vectors (apart from length) look about the same.

Consider the problem of trying to fit a fifteenth degree polynomial through 16 data points using the function `Fit[ ]`.

Start with the data points.

```
In[1]:= datapairs = Table[temp = 10. + i/20;
                         {temp, Sin[temp]},
                         {i, 16}]

Out[1]= {{10.05, -0.585277}, {10.1, -0.625071},
  {10.15, -0.663302}, {10.2, -0.699875}, {10.25, -0.734698},
  {10.3, -0.767686}, {10.35, -0.798754}, {10.4, -0.827826},
  {10.45, -0.854829}, {10.5, -0.879696}, {10.55, -0.902363},
  {10.6, -0.922775}, {10.65, -0.940881}, {10.7, -0.956635},
  {10.75, -0.969998}, {10.8, -0.980936}}
```

Define a basis for the problem.

```
In[2]:= basis = Table[x^i, {i,0,15}]

                 2   3   4   5   6   7   8   9   10   11   12
Out[2]= {1, x, x , x , x , x , x , x , x , x , x  , x  , x  ,

   13   14   15
  x  , x  , x  }
```

Find the best fit polynomial. Since there are 16 degrees of freedom in a degree 15 polynomial this should be an interpolating polynomial.

```
In[3]:= poly = Fit[datapairs, basis, x]

                     -16              -15                -15  2
Out[3]= 1.38382 10    + 9.67013 10    x + 7.85248 10    x  +

             -14  3              -13  4              -12  5
  5.65694 10    x  + 3.92338 10    x  + 2.59625 10    x  +

            -11  6              -11  7             -10  8
  1.6187 10    x  + 9.33312 10    x  + 4.8322 10    x  +

             -9  9              -9  10            -8  11
  2.13438 10   x  + 7.24375 10   x   + 1.4008 10   x   -

             -9  12             -10  13              -11  14
  5.69789 10   x   + 8.01711 10    x   - 4.91889 10    x   +

            -12  15
  1.1237 10    x
```

Notice that the errors are quite substantial. If we had used `InterpolatingPolynomial[ ]`, the errors would have been *much* smaller.

```
In[4]:= Map[(#[[2]] - (poly /. x->#[[1]]))&, datapairs]

Out[4]= {0.0000214232, -0.0000157863, -0.0000208176,

              -6            -6
  -9.95129 10  , 4.65817 10  , 0.000015065, 0.0000174084,

                          -7
  0.0000115397, 3.56274 10  , -0.0000112094, -0.0000176011,

                           -6
  -0.0000145107, -1.27846 10  , 0.0000161056, 0.0000213206,
  -0.0000167346}
```

Here again the problem is that the basis for the space we are considering is not orthogonal. In fact some of the basis functions, apart from scaling, are almost indistinguishable over the interval from 10 to 11. The plots of x^{14} and $5.2892028x^{13} + 0.047235340x^{15}$ are so similar as to appear as a single curve on the interval.

If we want to be precise, the concept of orthogonality assumes an inner product on the space, and in this particular example the inner product is a discrete one. We don't need exact orthogonality; orthogonality is just the ideal. We only need each of the basis functions to be quite different from each of the others. Rather than use basis functions that are orthogonal with respect to the inherent inner product[5], we can use basis functions that are orthogonal with respect to some other, similar inner product. In fact, just shifting the powers of x and using $(x-10)^k$ as the basis functions would help a great deal.

The Chebyshev polynomials $T_n(x)$ are orthogonal with respect to both a discrete and a continuous inner product over the interval $-1 \le x \le 1$. If we map the interval $10 \le x \le 11$ onto the interval $-1 \le x \le 1$ in a linear fashion, we can use the Chebyshev polynomials as a basis.

Define the basis functions in a way that they will evaluate only when `t` is a number.

```
In[5]:= cheby[i_,t_?NumberQ] := ChebyshevT[i, t]
```

```
In[6]:= basis = Table[cheby[i, t], {i, 0, 15}];
```

Map the data onto the interval $-1 \le x \le 1$.

```
In[7]:= mappeddata = datapairs /. {x_, y_} -> {2x - 21, y}

Out[7]= {{-0.9, -0.585277}, {-0.8, -0.625071},
  {-0.7, -0.663302}, {-0.6, -0.699875}, {-0.5, -0.734698},
  {-0.4, -0.767686}, {-0.3, -0.798754}, {-0.2, -0.827826},
  {-0.1, -0.854829}, {0., -0.879696}, {0.1, -0.902363},
  {0.2, -0.922775}, {0.3, -0.940881}, {0.4, -0.956635},
  {0.5, -0.969998}, {0.6, -0.980936}}
```

Express the interpolating polynomial in terms of our `basis`.

```
In[8]:= poly = Fit[mappeddata,basis,t]

Out[8]= -0.825568 cheby[0, t] - 0.230415 cheby[1, t] +
  0.0538445 cheby[2, t] + 0.0024383 cheby[3, t] -
                                       -6
  0.000282799 cheby[4, t] - 7.6596 10    cheby[5, t] +
             -7                        -8
  5.91287 10   cheby[6, t] + 1.13503 10   cheby[7, t] -
              -10                        -11
  6.94334 10    cheby[8, t] - 2.89948 10    cheby[9, t] -
              -12                         -12
  2.49539 10    cheby[10, t] - 6.29072 10    cheby[11, t] -
              -12                         -13
  4.28057 10    cheby[12, t] - 9.57091 10    cheby[13, t] -
              -13                        -14
  1.52079 10    cheby[14, t] - 5.0457 10    cheby[15, t]
```

The errors are much smaller.

```
In[9]:= Map[(#[[2]] - (poly /. t->(2 #[[1]] - 21)))&, datapairs]

                      -11            -13              -11
Out[9]= {7.67446 10   , -1.7486 10   , -3.05775 10   ,
              -12            -11            -10            -10
  4.77585 10   , 8.13334 10   , 1.69675 10   , 2.43012 10   ,
             -10            -10            -10             -11
  2.7552 10   , 2.45518 10   , 1.45017 10   , -1.15842 10   ,
               -10             -10            -10
  -1.87923 10   , -3.37655 10   , -4.2214 10   ,
               -10             -10
  -4.23009 10   , -3.41885 10   }
```

[5]In this particular problem we could actually find basis functions that are orthogonal with respect to the inner product. In other problems it may be more difficult.

This idea is implemented in a more robust way in the package `NumericalMath`PolynomialFit``. `PolynomialFit[ ]` takes two arguments, which correspond to the first two arguments of `Fit[ ]`. However, the second argument to `PolynomialFit[ ]` is simply an integer that specifies the degree of the polynomial rather than a list of basis functions. `PolynomialFit[ ]` returns a pure function so a variable is not needed as it is in `Fit[ ]`. In addition, `PolynomialFit[ ]` is not designed for multivariate fitting, although it could be.

Load the package.

```
In[10]:= << NumericalMath`PolynomialFit`
```

```
In[11]:= poly = PolynomialFit[datapairs, 15]

Out[11]= FittingPolynomial[<>, 15]
```

The errors are smaller yet.

```
In[12]:= {xx, yy} = Transpose[datapairs]; Map[poly, xx]-yy

                      -12             -12               -13
Out[12]= {2.97706 10   , 1.91647 10   , -1.95954 10   ,

                 -12              -12              -13
  -1.60771 10   , -1.50946 10   , -5.49671 10   ,

                -13             -14              -13              -13
  1.36779 10   , 1.92069 10   , -4.94382 10   , -5.95524 10   ,

                -14             -12             -12            -13
  8.72635 10   , 1.07725 10   , 1.42619 10   , 5.7232 10   ,

                 -12              -12
  -1.00575 10   , -2.11753 10   }
```

We could expand the polynomial to see what it is.

```
In[13]:= Expand[poly[x]]

                    11            11                11  2
Out[13]= 1.21499 10   - 1.749 10   x + 1.1749 10   x  -

              10  3              10  4             9  5
  4.88567 10   x  + 1.40647 10   x  - 2.9691 10  x  +

              8  6             7  7              6  8
  4.74823 10  x  - 5.8576 10  x  + 5.62017 10  x  -

           9             10             11             12
  419393. x  + 24142.1 x   - 1052.79 x   + 33.6661 x   -

             13               14                  15
  0.745301 x   + 0.0102136 x   - 0.0000653164 x
```

But the results would be even worse than the first attempt. The function `InterpolatingPolynomial[ ]` can be similarly abused; its result is left in an awkward-looking form for reasons of numerical stability.

```
In[14]:= (% /. x -> xx) - yy

Out[14]= {-0.0397227, -0.374929, 0.288302, 0.324875,
  -0.0153016, 0.267686, -0.201246, 0.327826, 0.229829,
  -0.370304, 0.152363, -0.202225, 0.315881, -0.168365,
  0.594998, 0.355936}
```