Floating-Point Arithmetic

Precision and Accuracy

With Mathematica

Alkis Akritas

University of Kansas
Dept of Computer Science
415 Snow Hall
Lawrence KS 66045
USA

akritas@eecs.ukans.edu

Introduction

In performing numerical computations on a computer, we are faced with the problem of representing the *infinite* set of real numbers within a computer of finite memory and of given word length; therefore, certain compromises have to be made. For example, `Sqrt[2]` represents some number which is *approximately* `1.41421`. If we want to represent it exactly there is little choice but to leave it as `Sqrt[2]`. For arithmetic purposes this is rather useless, but to say that `Sqrt[2]` is equal to `1.41421` (or, any other *finite* decimal expansion) is simply wrong. What is usually done is to work with approximations and accept the fact that the end result is also an approximation.

The most widely implemented solution in Numerical Analysis is to approximate the real numbers using the *finite* set of *floating-point* numbers. *Mathematica* has *two* kinds of floating-point numbers: *machine-precision* numbers and *arbitrary-precision* numbers. (The precision of a number is defined below.) Machine-precision numbers contain a fixed number of digits, and their precision remains unchanged throughout computations. On the other hand, arbitrary-pecision numbers can contain any number of digits, and their precision is adjusted during computations; as we will see later, this implementation is based on the ideas of interval arithmetic.

Machine-precision numbers make direct use of the numerical capabilities of the computer system we are using. Therefore, the arithmetic operations are hardware implemented and quite fast. On the other hand, the arithmetic operations for arbitrary-pecision numbers are software implemented and, hence, much slower.

Typically, machine-precision numbers in *Mathematica* are represented as "double-precision floating-point numbers" in our computer systems. For example, this computer has

```
$MachinePrecision

19
```

which means that we can represent numbers with at most 19 digits. As a rule of thumb, 10 digits can be stored in a computer word with 32 bits (single-precision), so this computer is using 64 bits (double-precision) for the representation of machine-numbers.

When we enter an approximate real number, *Mathematica* has to decide whether to treat it as machine precision or arbitrary precision. If we give less than `$MachinePrecision` digits, then the number is treated as machine precision, and if we give more digits, then the number is treated as arbitrary precision. For example,

```
{MachineNumberQ[2.0],
    MachineNumberQ[2.00000000000000000000]}
```

```
{True, False}
```

In the case of machine-precision, *Mathematica* assumes that all digits after the ones we entered are zero, whereas in the case of arbitrary-precision *Mathematica* assigns the number a precision equal to the number of digits we entered.

```
{Precision[2.0], Precision[2.00000000000000000000]}
```

```
{19, 21}
```

It should be emphasized that even if we typed 2.0 its precision is not 1 or 2 but 19. Machine numbers have a minimum precision, the so-called *machine* precision which is equal to `$MachinePrecision`. As we stated, it is for these numbers that *Mathematica* uses the computer hardware to perform approximate calculations. Since it is impractical to keep track of their precision, it is assumed that it is always the same.

In *Mathematica,* arithmetic with machine numbers generally results in a machine number unless the result is too large or too small. If the result cannot be represented as a machine number, it is converted to an arbitrary-precision number with `$MachinePrecision` digits. If no machine numbers are involved, arithmetic with arbitrary-precision numbers results in an arbitrary-precision number, even if the result has fewer than `$MachinePrecision` digits.

The *Mathematica* function `N[]` numerically evaluates an expression to give a result that is only approximately correct; that is, the result is a floating-point number. Consider for example `Pi`.

```
MachineNumberQ[ Pi ]
```

```
False
```

Applying the function `N[]` to `Pi`, we convert it to a machine number with value `3.14159`.

```
{N[Pi], MachineNumberQ[ N[Pi] ]}
```

```
{3.14159, True}
```

Here is a word of caution. Unless we explicitly ask for a specified number of digits, *Mathematica* displays only six even though it has computed more, as can be seen from the input form; and, it is important to remember that `N[Pi]` and `3.14159` are *not equal*.

```
{InputForm[ N[Pi] ], Equal[ N[Pi],3.14159 ]}
```

```
{3.141592653589793238, False}
```

Even though the default precision in *Mathematica* is six digits, we can use the function `N[]` to obtain a higher precision floating-point representation for `Pi` that is *not* a machine number.

```
{N[Pi, 20], MachineNumberQ[ N[Pi, 20] ]}
```

```
{3.14159265358979323846, False}
```

The floating-point number `N[Pi, 20]` is not a machine number because its precision is higher than 19, the `$MachinePrecision` of this computer.

At times, however, the function `N[]` does not provide the requested precision. Below we ask for a precision of 30 digits but we only receive `$MachinePrecision` digits.

```
{N[ Sqrt[2.], 30 ],
    Precision[ N[ Sqrt[2.], 30 ] ]}
```

```
{1.414213562373095049, 19}
```

This happens because, as we have seen in the last section, 2. has a precision of 19 (`$MachinePrecision`) digits. Since the starting point of the calculation is precise to only 19 places, asking for 30 places is not appropriate. If we want high-precision answers with `N[]`, we have to start either with exact integers (of infinite precision) or with approximate numbers with enough places of precision:

```
{N[ Sqrt[2], 30 ],
    N[Sqrt[   2.00000000000000000000000000000],30
    ]}
```

```
{1.41421356237309504880168872421,
  1.41421356237309504880168872421}
```

Another option is to use the function `SetPrecision[]` which creates a number of a specified precision (within limits), padding if necessary with binary zeros. (The function `SetAccuracy[]` also exists.)

```
SetPrecision[ Sqrt[2.], 30 ]
```

```
1.41421356237309504876378807303
```

However, despite *Mathematica*'s statements, that

```
{Precision[%], Accuracy[%]}
```

```
{30, 30}
```

the accuracy of the result is `$MachinePrecision` digits. The reason for it is that when we enter `Sqrt[2.]` *Mathematica* representes it as a machine-precision number, and then pads the number with binary zeros until the requested precision is achieved. However, `Sqrt[2.]` *cannot* retain the extra digits of information (in the 64 bits of its double-precision representation), and hence its value is not exactly `Sqrt[2.]`. So the number is within `$MachinePrecision` digits of the true value, even though it is expressed in 30 digits.

Precision and Accuracy

We begin with various definitions:

$$\textit{error} \overset{\text{def}}{=} \textit{approximate value - true value}$$

which is consistent with popular use. In general we would like our errors to be small, but what does "small" mean? It depends on how small the values are that we are approximating, and, hence, we want the error to be *relatively* small. That is we need the

$$\textit{relative error} \overset{\text{def}}{=} \frac{\text{error}}{\text{true value}}$$

Because errors are usually very small it is convenient to introduce logarithmic measures of absolute and relative error. It is with the help of these measures that *Mathematica* defines precision and accuracy. Let us define

$$|\textit{error}| = 10^{-d} \quad \text{and} \quad |\textit{relative error}| = 10^{-\delta}$$

Solving for d we obtain

$$\textit{accuracy} \overset{\text{def}}{=} -\log_{10}|\textit{error}|$$

which means that, if *d* is an integer, the approximation has *d* accurate *decimal* places, or, equivalently, that the *d*th digit *to the right of the decimal point* is off by one unit. On the other hand, solving for δ we obtain

$$precision \overset{\text{def}}{=} -\log_{10}|relative\ error|$$

which means that, if δ is an integer, the approximation has δ accurate digits, or, equivalently, that the δth digit *counting from the first nonzero digit* is off by one unit. Note that

precision - accuracy

gives the number of digits to the *left* of the decimal point; this will be a negative number in case there are no digits to the left of the decimal point *and* there are some zeros after it before we reach the significant digits.

To illustrate these ideas consider

```
        true value = 1.61803
 approximate value = 1.61924.
```

A simple calculation indicates that the precision is 3.12 digits and that the accuracy is 2.91 digits. This agrees with the observation that there are three accurate digits and two accurate decimal digits.

Mathematica functions `Precision[]` and `Accuracy[]` give us precisely this information. For complex numbers the accuracy and precision of the real and imaginary parts is maintained separately. `Precision[]` and `Accuracy[]` report the *minimum* of the two precisions and accuracies, and in all cases the values obtained are rounded to the nearest number of decimal digits.

Let us consider the function `pa` for obtaining both precision and accuracy.

```
pa[r_?NumberQ] :=
    {Precision[r], Accuracy[r]}
```

Then for the product `1000 Sqrt[2.]`, for which we have defined its precision to 30, we have:

```
pa[SetPrecision[1000 Sqrt[2.], 30]]
```

```
{30, 27}
```

From the above we see that whereas the precision is as expected, the accuracy, *due to rounding*, is not the desired one, because now the difference `precision - accuracy` gives us only three, instead of four, digits to the left of the decimal point. Switching off rounding we obtain

```
SetPrecision[Round, False];
```

```
pa[SetPrecision[1000 Sqrt[2.], 30]]
```

```
{30., 26.8495}
```

which agrees with our observation that there should be 30 accurate digits and 26 accurate decimal digits.

Note that precision can be lost in a calculation. Below we start with 30 places of accuracy, and we obtain a result accurate to only 10.

```
N[10^20 + Pi, 30] - 10^20
```

```
3.1415926536
```

However, precision can also increase as is seen below

```
{Precision[ N[Pi, 30] ],
            Precision[ N[Pi, 30] + 10^20 ]}
```

```
{30., 49.5029}
```

Machine-Precision Floating-point Numbers

In this section we will deal exclusively with machine-precision floating-point numbers which, for short, will be refered to either as machine numbers or as floating-point numbers.

A set F of floating-point numbers is characterized by three parameters: the number *base* β, the *precision* n which is the number of digits in the *mantissa*, and the *exponent range* [L, U].

Each floating-point number f in F can be represented as

$$f = \pm\left(\frac{d_1}{\beta} + \frac{d_2}{\beta^2} + \cdots + \frac{d_n}{\beta^n}\right)\beta^e$$

where the digits in the mantissa d_i, i = 1, 2, ..., n satisfy the inequality $0 \le d_i \le \beta - 1$ and $L <= e \le U$; if we also require $d_1 \ne 0$ for all f in F, $f \ne 0$, we have the *normalized* floating-point numbers.

For example, the IEEE standard requires: β = 2, n = 24 and exponent range [-126, 127] for single-precision and β = 2, n = 53 and exponent range [-1022, 1023] for double-precision. In the case of single-precision the machine numbers according to this standard are

$\pm 2^e (b_1 b_2 b_3 \cdots b_{24})_2$, $-126 \le e \le 127$,
± Infinity,
NaN (Not-a-Number).

Observe that the set F is *not* continuous or even an infinite set. There are exactly

$$2 (\beta - 1)\beta^{n-1} (U - L + 1) + 1$$

normalized floating point numbers, including zero, in F; moreover, these numbers are *not* equally spaced throughout their range but only between powers of β. An example that we will consider below will further clarify these notions.

Because of the above, it is possible that given f_1 and f_2 in F, their sum (or product) will *not* be in F and in that case it will have to be approximated by the closest floating-point number; as we will see there are various ways for this approximation depending on the rounding rule. The difference between the true and the approximated sum (or product) is the *round-off* error. It should be kept in mind that the operations of addition and multiplication in F are *not* associative and the distributive law fails. Therefore, when dealing with floating-point numbers we have to watch out not only for computational errors, but also for the *order* in which the operations are performed.

To learn about the basic ideas of computer floating-point arithmetic we use the *Mathematica* package `NumericalMath`ComputerArithmetic``:

```
Needs["NumericalMath`ComputerArithmetic`"]
```

which contains the following

```
Names["NumericalMath`ComputerArithmetic`*"]
```

```
{Arithmetic, ComputerNumber, ExponentRange, IdealDivide,
  MixedMode, NaN, RoundingRule, RoundToEven,
  RoundToInfinity, SetArithmetic, Truncation}
```

Before we proceed note that if we want to find out more about a certain function of the package, say `Arithmetic[]`, we can type `??Arithmetic`; but the cryptic output refers to variables of the form `NumericalMath`ComputerArithmetic`Private`$**` and it is very hard to read. To obtain a simpler version of the function definition (code) we have to change into the appropriate context. That is we first type in

```
Begin["NumericalMath`ComputerArithmetic`Private`"]
```

```
NumericalMath`ComputerArithmetic`Private`
```

examine `Arithmetic[]` or any other functions we want, and when we are done we type

```
End[]
```

```
NumericalMath`ComputerArithmetic`Private`
```

With the package now loaded, and after examining various functions, we check its default arithmetic

```
Arithmetic[]
```

```
{4, 10, RoundingRule -> RoundToEven,
  ExponentRange -> {-50, 50}, MixedMode -> False,
  IdealDivide -> False}
```

and see that it is four digits to base 10 with a rounding rule `RoundToEven` (that is, round to the nearest, and in case of ties, round to the one with an even m a n t i s s a; other rounding options are: RoundToInfinity and Truncation, where RoundToInfinity means round to the nearest and in case of ties, away from zero). Given the exponent range only numbers between 10^{-50} and .9999 10^{50} are allowed. Mixed-mode arithmetic is not allowed and division is not done with correct rounding.

We want to deal with a set F that has fewer points than the one above and so we define a new arithmetic with three digits to base 2 and a much smaller exponent range

```
SetArithmetic[3, 2, ExponentRange ->
    {-2, 2}]
```

```
{3, 2, RoundingRule -> RoundToEven,
  ExponentRange -> {-2, 2}, MixedMode -> False,
  IdealDivide -> False}
```

The exponent range {-2, 2} in *Mathematica* is equivalent to the exponent ranges {-1, 2}, when the mantissa is < 1, and {-2, 1} when the mantissa is >= 1 (see the definitions above). In all cases the new set F has 33 normalized floating-point numbers between -7/2 and 7/2.

We will examine the positive floating-point numbers of F with the help of `ComputerNumber[]`; this is a data object that represents a computer number, and its default print format is a simple number in the base specified by `SetArithmetic[].`

Starting with the smallest number in our arithmetic, 1/4, we see that it is represented as

```
InputForm[ComputerNumber[1/4]]
```

```
ComputerNumber[1, 4, -4, 1/4, 0.2500000000000000000000]
```

where the sign is 1, the mantissa 4, the exponent -4, the value is 1/4 and the last entry is a high-precision representation of the value. In a real computer only the first three entries show up. Since the mantissa is > 1 we see that, following the IEEE standard, the positive floating-point numbers of F can be represented as

$$2^e(1.b_2\, b_3)_2 \text{ or } 2^{e-2}(1\, b_2\, b_3)_2, \quad -2 \le e \le 1$$

or equivalently as

$$2^p m,$$

where $-4 \le p \le -1$ and $2^2 \le m \le 2^3 - 1$.

The smallest machine number of our arithmetic, 1/4, corresponds to m = 4 (= $1b_2\,b_3$ = 100) and p = -4. The other numbers corresponding to p = -4 are obtained by setting m = 5, 6, and 7 which is equivalent to varying the values of $b_2\,b_3$ in a systematic way. Thus, for p = -4 the largest number is 7/16 and is obtained when m = 7 (= $1b_2\,b_3$ = 111). Note that no machine numbers exist between 0 and 1/4, and that the values of m are the same for each value of the exponent p = -3, -2, -1.

The 16 positive machine numbers of our arithmetic can be easily obtained in binary form:

```
binaryMachineNumbers =
    Table[ ComputerNumber[2^p m],
        {p, -4, -1}, {m, 4, 7}]
```

```
{{0.01 , 0.0101 , 0.011 , 0.0111 },
      2         2        2         2
  {0.1 , 0.101 , 0.11 , 0.111 },
      2       2      2       2
  {1. , 1.01 , 1.1 , 1.11 }, {10. , 10.1 , 11. , 11.1 }}
     2      2     2      2       2       2      2       2
```

and likewise in decimal:

```
machineNumbers = Map[ Normal,
             binaryMachineNumbers // Flatten ]
```

```
 1   5   3   7   1  5  3  7      5  3  7      5      7
{-, --, -, --, -, -, -, -, 1, -, -, -, 2, -, 3, -}
 4  16  8  16  2  8  4  8      4  2  4      2      2
```

Normal[] converts a computer number back to an ordinary rational number with exactly the same value.

To see how the 16 positive floating-point numbers are distributed on the x-axis, define the discrete equivalent to derivative, that is the difference function

```
diff[x_List] := Drop[x, 1] - Drop[x, -1]
```

and apply it to machineNumbers. We obtain

```
diff[machineNumbers]
```

```
 1   1   1   1   1  1  1  1  1  1  1  1  1  1  1
{--, --, --, --, -, -, -, -, -, -, -, -, -, -, -}
 16  16  16  16  8  8  8  8  4  4  4  4  2  2  2
```

from which we see that the floating-point numbers in F are equally spaced for a given exponent. Note also that the difference (distance) between adjacent machine numbers doubles every time we change exponents.

We can also see a diagram of the distribution of the positive floating-point numbers of our arithmetic. Numbers corresponding to the same exponent value will have the same point size, and sizes will increase with the exponent value.

To achieve our goal, we have to first turn the machineNumbers into a list of points in the (x, y) plane; that is, we have to append to each number its y-coordinate which is 0.

```
pList = Map[ {#, 0}&, machineNumbers]
```

```
   1         5          3         7          1         5         3
{{-, 0}, {--, 0}, {-, 0}, {--, 0}, {-, 0}, {-, 0}, {-, 0},
   4        16         8        16         2         8         4
   7                   5         3         7                   5
  {-, 0}, {1, 0}, {-, 0}, {-, 0}, {-, 0}, {2, 0}, {-, 0},
   8                   4         2         4                   2
            7
  {3, 0}, {-, 0}}
            2
```

We then create a list of sizes for each point as described above:

```
sList = Table[ i, {i, 2, 5}, {4} ] //Flatten
```

```
{2, 2, 2, 2, 3, 3, 3, 3, 4, 4, 4, 4, 5, 5, 5, 5}
```

Transposing the two lists we obtain the list of points-sizes

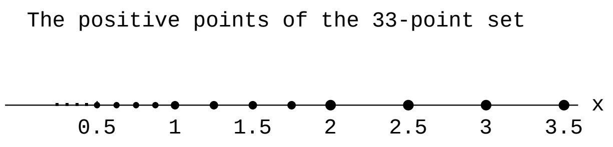
```
psList = Transpose[{pList, sList}];
```

Next, we write a function to make points of a given size

```
makePoints[{pList_, size_?Positive}]:= ({
    AbsolutePointSize[size], Map[Point, {pList}] })
```

and map it on the points-sizes list

```
Show[ Graphics[ Map[makePoints, psList] ],
      Ticks -> {{0, .5, 1, 1.5, 2, 2.5, 3, 3.5},
                        Automatic},
      Axes -> {True, False}, AxesLabel -> {"x", " "},
      Epilog -> {Text["The positive points of the 33-point set",
        {0.05, 0.25}, {-1, 0}]}  ]
```



to obtain the desired graph of the distribution of the points. As mentioned above, points of the same size correspond to floating-point numbers having the same exponent, and, therefore, are equally spaced.

Continuing with our example arithmetic, it should be clearly understood that, as far as the computer is concerned, only these 16 machine numbers exist on the positive x axis. Every other number from zero to infinity, provided it is not too small or too large, has to be approximated and represented by one of these 16 machine numbers.

Note that we cannot represent numbers sufficiently smaller than 1/4

```
ComputerNumber[1/8]
```

```
ComputerNumber::undflw:
   Underflow occurred in computation.  The exponent is -2.
```

```
NaN
```

that is, 1/8 is `Not-a-Number` for this arithmetic; and the same is true for numbers sufficiently greater than 7/2

```
ComputerNumber[4]
```

```
ComputerNumber::ovrflw:
   Overflow occurred in computation.  The exponent is 3.
```

```
NaN
```

To understand how numbers between 1/4 and 7/2 are represented by the 16 floating-point numbers consider various numbers between, say, 1/4 and 5/16. The number 43/160 (= 1/4 + 3/160) is approximated by 1/4 which is the nearest machine number

```
Normal[ComputerNumber[1/4 + 3/10 1/16]]
```

```
1
-
4
```

On the contrary, the number 47/160 is approximated by 5/16, because this is now the nearest machine number

```
Normal[ComputerNumber[1/4 + 7/10 1/16]]
```

```
5
--
16
```

Finally the number 9/32 which lies exactly in the center between 1/4 and 5/16 is approximated by 1/4 due to the fact that the mantissa of 1/4 is an even number; recall that our rounding rule is RoundToEven.

```
Normal[ComputerNumber[1/4 + 1/2 1/16]]
```

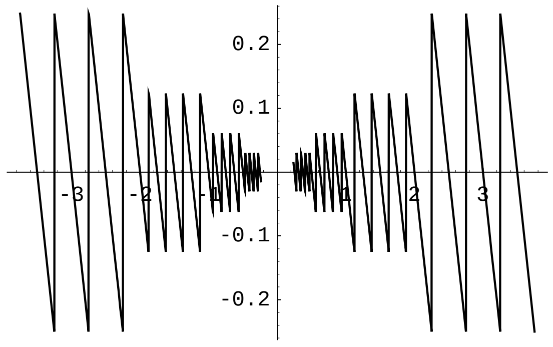
```
1
-
4
```

Therefore, every time we convert a number x to its floating-point equivalent, ComputerNumber[x], an error incurres which is equal to Normal[ComputerNumber[x]] - x. This error is 0 only in case x is a machine number.

We can easily plot the error incurred in the computer representation of each number from -4 to 4, using the machine numbers of our current arithmetic.

```
Plot[Normal[ComputerNumber[x]] - x, {x, -4, 4},
    PlotPoints -> 141,
    PlotLabel -> "Representation errors with RoundToEven"]
```

Representation errors with RoundToEven



The vertical lines in the figure above correspond to discontinuity jumps.

>From the discussion above it is clear that numbers on a computer comprise a discrete set, and, since there are gaps between the numbers, when we do arithmetic the result often is not representable, i.e. it is not a machine number. When the result falls between two machine numbers the best that can be done is to use the closest machine number as the result instead of the correct result.

We should also keep in mind that it is not just numbers like Pi or Sqrt[2], with an infinite number of digits in their decimal expansions, that have to be approximated by machine numbers. There exist numbers like 1/10 = 0.1 or 88/100 = 0.88, with an infinite number of (repeating) digits in their *binary* expansions which also have to be approximated by machine numbers.

```
BaseForm[ N[1/10, 30], 2]
```

```
0.00011001100110011001100110011001100110011001100110011001\
   1001100110011001100110011001100110011001101
                                              2
```

To demonstrate that associativity does not hold in floating-point arithmetic consider in our 33 point set F the sum

```
5/4 + 3/8 + 3/8.
```

The result we obtain depends on the order in which we do the additions and on the rounding rule. Trying the existing rounding rule with the two possible orders we obtain:

```
{Normal[ComputerNumber[5/4 + 3/8] +
                    ComputerNumber[3/8]],
    Normal[ComputerNumber[5/4] +
                    ComputerNumber[3/8 + 3/8]]}
```

```
{2, 2}
```

i.e., everything works as expected. However, changing the rounding rule to `Truncation`

```
SetArithmetic[3, 2, ExponentRange -> {-2, 2},
    RoundingRule -> Truncation];
```

we obtain

```
{Normal[ComputerNumber[5/4 + 3/8] +
                        ComputerNumber[3/8]],
    Normal[ComputerNumber[5/4] +
                        ComputerNumber[3/8 + 3/8]]}
```

$\{\frac{7}{4}, 2\}$

which implies that (5/4 + 3/8) + 3/8 = 7/4 does *not* equal 5/4 + (3/8 + 3/8) = 2.

We can examine the floating-point arithmetic on this computer using the *Mathematica* package `Numeri-calMath`Microscope``

```
Needs["NumericalMath`Microscope`"]
```

which contains the following

```
Names["NumericalMath`Microscope`*"]
```

```
{MachineError, Microscope, MicroscopicError, Ulp, Ulps}
```

The difference between two consecutive machine numbers is called an *ulp* (*u*nit in the *l*ast *p*lace) and, as we have seen above, its size varies depending on where we are in the set of machine numbers, and changes at powers of 2. Define `$MachineEpsilon` the difference between 1.0 and the next larger machine number; then between 1 and 2 an ulp is equal to `$MachineEpsilon`; between 2 and 4 it is equal to `2 $Machine-Epsilon` etc. (The values below are computer depended.)

```
$MachineEpsilon
```

$1.0842\ 10^{-19}$

```
{Ulp[1.5], Ulp[2], Ulp[4], Ulp[8]}
```

$\{1.0842\ 10^{-19}, 1.0842\ 10^{-19}, 2.1684\ 10^{-19}, 4.33681\ 10^{-19}\}$

Ideally no function should ever return a result with error exceeding half an ulp, since the distance from the true result to the nearest machine number is always less than half an ulp, and the worst case is when it is exactly in the middle of two machine numbers.

Below we examine the behavior of the square root function on this computer using the functions of `Numeri` `calMath`Microscope``. As a first step, let us look at the combined error of rounding 29Pi/30 to the nearest machine number and then taking the square root.

```
MachineError[ Sqrt[x], x -> 29 Pi/30 ]

0.719947 Ulps
```

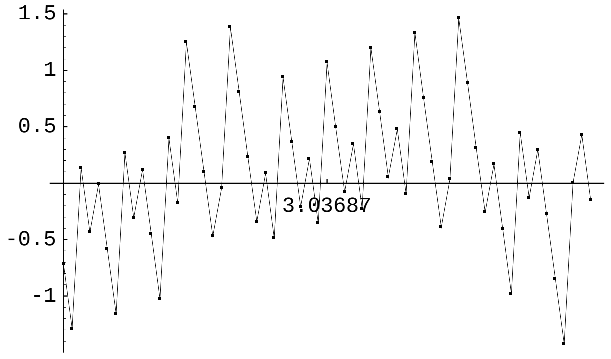
Examining the error of just the `Sqrt[]` function we obtain

```
MachineError[ Sqrt[x], x -> N[ 29 Pi/30] ]

1.06981 Ulps
```

and it *seems* that the `Sqrt[]` function does not behave ideally on this computer (in the sense described above) since its error is greater than half an ulp. Actually, from 30 ulps to the left of 29Pi/30 to 30 ulps to the right of 29Pi/30 we have the following diagram of the errors:

```
MicroscopicError[ Sqrt[x], {x, N[29 Pi/30]} ]
```



```
-Graphics-
```

from which we see that at the point 15 Ulps to the right of 29Pi/30 we have a very big error

```
 MachineError[Sqrt[x],x -> N[29 Pi/30] +
                              15 Ulp[29 Pi/30]]

1.46229 Ulps
```

Does `Sqrt[]` really behave badly on this computer or are these errors due to the rounding error incurred in converting all these numbers containing `Pi` to machine numbers? The truth is that we should be careful with our judgement. As far as the computer is concerned `Pi` does not exist, and all `Sqrt[]` does is to map a machine number approximating `Pi` into another machine number. Therefore, the function should not be blamed for the rounding error incurred in converting `Pi` to a machine number.

We close this section with the remark that comparison operators in *Mathematica* are inexact. That is, if two numbers are very close together, *Mathematica* will consider them equal.

```
{x = 1.0, y = 1 + 2 $MachineEpsilon}

{1., 1.}

x == y

True
```

Note also that

```
x < y

False
```

Therefore, the only way to obtain exact comparisons is to subtract the two numbers we are trying to compare.

```
x - y < 0
```

```
True
```