The apnum package: Arbitrary precision numbers implemented in TEX macros

Petr Olšák

TEX doesn't provide a comfortable environment for calculations at the primitive level. There are the well-known commands `\advance`, `\multiply` and `\divide` but dividing two decimal numbers with these commands is a somewhat complicated task for macro programmers. The additional ε-TEX primitives `\numexpr` and `\dimexpr` do not make it easier. Of course, various LATEX packages exist for more comfortable numerical calculations. None of them satisfied my needs so I decided to create my own solution: the `apnum.tex` package (`http://www.ctan.org/pkg/apnum`). You can set an arbitrary precision for the calculation when using this package. You can do addition, multiplication, division, power, square root, and evaluation of common functions sin, cos, tan, arcsin, arccos, arctan, exp and ln. The result is calculated with `\apFRAC` decimal positions after the decimal point. You can set this register to an arbitrary value. Of course, if you need thousands of decimal digits then you must wait a while. Nevertheless, optimization techniques were used when implementing algorithms.

The documentation is in the file `apnum.pdf`. It is not only the user-level documentation, but also detailed technical documentation is included. You can find the description of all internal macros and all the numerical algorithms used.

The expression scanner

After `\input apnum` in your document, you can use the macro `\evaldef`⟨*sequence*⟩`{`⟨*expression*⟩`}`. It makes comfortable calculation available. The ⟨*expression*⟩ can include binary operators `+`, `-`, `*`, `/` and `^` with the usual precedence. The operands are "numbers". Users can use parentheses `()` as usual. The result is stored to the ⟨*sequence*⟩ as a "literal macro". Examples:

```
\evaldef\A {2+4*(3+7)}
   % the macro \A is the result, 42
\evaldef\B {\the\pageno * \A}
   % \B is 42 times page number
\evaldef\C {123456789 * -123456789123456789}
   % \C is -15241578765432099750190521
\evaldef\D {1.23456789 + 12345678.9 - \A}
   % \D is 12345596.13456789
\evaldef\X {1/3}
   % \X is .33333333333333333333
```

The result of division doesn't have absolute precision; the number of digits after the decimal point is limited by the value of `\apFRAC`, which is 20 by default. Absolute precision is implemented when `+`, `-`, `*` and `^` operators are used. When using `/` or evaluating math functions like $\sin x$, only `\apFRAC` digits are calculated after the decimal point.

The operands in the ⟨*expression*⟩ are most simply numbers in the format

⟨*sign*⟩⟨*digits*⟩`.`⟨*digits*⟩

where optional ⟨*sign*⟩ is a sequence of `+` and/or `-` characters. A nonzero number is treated as negative if and only if there is an odd number of `-` signs. The first group or second group of decimal ⟨*digits*⟩ (but not both) can be empty. The decimal point is optional if the second group of ⟨*digits*⟩ is empty.

Alternatively, you can specify an operand in scientific notation in the format

⟨*sign*⟩⟨*digits*⟩`.`⟨*digits*⟩`E`⟨*e-sign*⟩⟨*digits*⟩

The sequence before `E` determines the mantissa and the sequence after `E` is the exponent. The ⟨*e-sign*⟩ is `+` or `-` or nothing. If you are using scientific notation of operands then the result (calculated by `\evaldef`) is usually in the same form. The reason is simple. If you want to calculate (for example) `3E+2000 * 5E+1300` then `apnum` will not waste time working with "full numbers" with a lot of digits (converted from scientific form) but calculates only `3*5=15` and the exponent of the result `3300` is appended. Much more information about scientific format is in the documentation of the `apnum` package.

The operands in the ⟨*expressions*⟩ can be any of the following:

- Numbers, as described above.
- `\the`⟨*register*⟩ or `\number`⟨*register*⟩. This allows accessing TEX register values.
- A macro which expands directly to a number. This allows working with "variables".
- A "function-like" macro which returns a value. This allows the implementation of functions. The identifier of a function-like macro can be followed by zero or more parameters, each of which must be enclosed in braces `{}`.

For instance, `\EXP` is a function-like macro. This macro has one parameter which is another (nested) ⟨*expression*⟩. The `\EXP` macro returns the value of the exponential function e^x where x is the given ⟨*expression*⟩. Example:

```
\def\X{.25}
\evaldef\A{\EXP{2*\X} - 1}
    % \A is the result of e^{2X} - 1
```

Users can define their own function-like macros; see the next section. The package apnum defines following function-like macros (with one parameter as nested ⟨*expression*⟩): \SQRT, \EXP, \LN, \SIN, \COS, \TAN, \ASIN, \ACOS, \ATAN. The meaning of these macros is clear from their names.

Note that you must use parentheses () for precedence settings in an ⟨*expression*⟩, but use braces {} as delimiters of parameters of function-like macros. The spaces in the ⟨*expression*⟩ are ignored. Example:

```
\def\A{20}
\evaldef\B{ 30*\SQRT{ \SIN{\PI/6} +
            1.12*\the\widowpenalty } / (4-\A) }
```

The evaluation of operators and function-like macros works at the main processor level of TEX. Unlike some comparable LATEX packages, the apnum package doesn't support calculations at the expansion processor level only. The reason is calculation speed optimization.. Moreover, I wanted it to be possible to use apnum in classical TEX, without ε-TEX primitives. And without the ε-TEX primitives then expansion-level calculation is very complicated. So, I rejected the expansion-only calculation. Another significant advantage of this decision is related to the possibility of creating function-like macros by users: they need not worry about main processor vs. expansion processor evaluation when creating their own function-like macros.

In my opinion, a skillful macro programmer doesn't require expansion-level calculation. He/she can use

```
\evaldef\V{⟨expression⟩}\edef\foo{...\V...}
```

instead of `\edef\foo{...`⟨*expression*⟩`...}` whenever he/she needs to do this.

There are some side effects of \evaldef processing:

- The value of the \apSIGN register. It is set to −1, 0, 1 according to whether the result is negative, zero or positive.
- The internal macro \OUT is a copy of the result.

Creating function-like macros

Let us start with creating our own function-like macros. We must follow two rules:

- The first token must be \relax after the first level of expansion. This is a signal that this is a function-like macro and not a normal numerical constant. The expression scanner creates a new TEX group and executes the macro in it.
- The function-like macro must define the macro \OUT as the result of processing, as a number, and the \apSIGN register must be set to the sign of the result. The expression scanner takes control again and uses these values as one operand in the ⟨*expression*⟩ currently being processed.

Several examples of function-like macros follow.

Hyperbolic sine (and inverse). There are some well-known mathematical functions not predefined in the apnum package. I believe that remembering the name of such a function is not markedly easier than remembering its natural definition, and the latter is much more useful and educational. So, I left such work to users. For example the hyperbolic sine can be defined by

```
\def\SINH#1{\relax % mandatory \relax
   \evaldef\myE{\EXP{#1}}%
   \evaldef\OUT{ (\myE - 1/\myE) / 2 }%
}
```

This corresponds to the formula

$$\sinh x = \frac{e^x - e^{-x}}{2}.$$

First, the mandatory \relax is given in the macro. Then the value e^x is saved in a temporary macro \myE. We need not worry about a name conflict (\myE being used elsewhere) because the macro is processed in the TEX group. The final \evaldef gives the desired result (including \apSIGN setting).

A reader may have another idea:

```
\def\SINH#1{\relax
  \evaldef\OUT{(\EXP{#1} - \EXP{-(#1)})/2}}
```

This implementation of \SINH also works, but it is not optimal because the slow calculation of \EXP is done twice. The internal ⟨*expression*⟩ #1 must also be evaluated twice.

Another example is the inverse of hyperbolic sine:

```
\def\ASINH#1{\relax
  \evaldef\X{#1}\LN{\X+\SQRT{\X^2+1}}}
```

The following identity is used here

$$\sinh^{-1} x = \ln\left(x + \sqrt{x^2+1}\right).$$

Here, we do not need to explicitly define the \OUT macro because \LN is another function-like macro, so it does this work.

Sine of argument in degrees. The default function-like macros \SIN, \COS and \TAN expect their argument in radians, meaning they have a period 2π. Sometimes, it is useful to use degrees instead of radians. There is a simple way to define

functions \SINdeg, \COSdeg and \TANdeg with argument in degrees:

```
\def\SINdeg#1{\relax \SIN{(\PI/180)*(#1)}}
\def\COSdeg#1{\relax \COS{(\PI/180)*(#1)}}
\def\TANdeg#1{\relax \TAN{(\PI/180)*(#1)}}
```

Note the parentheses around the #1 argument. This is because the argument may be an expression with (say) an addition.

We have another problem: the values in degrees are typically expressed in sexagesimal numeral system (degrees, minutes, seconds). Thus we create the function-like macro \DEG to take the value in sexagesimal notation and return a normal decimal number. Namely:

```
\DEG{12;30'45.756''}% means
  % 12 degrees, 30 minutes, 45.756 seconds
\evaldef\a{\DEG{12;30'45.756''}} % = 12.51271
\evaldef\a{\DEG{12;30'}}%        % = 12.5
\evaldef\a{\DEG{12}}%            % = 12
\evaldef\a{\DEG{12.5}}%          % = 12.5
```

The conversion is done only if a semicolon is used after the degrees value. On the other hand, a dot means the normal decimal dot in the number. The symbol for minutes ' and seconds '' at the end of the value is optional, so \DEG{12;30} also returns 12.5. The \DEG macro can be defined using this code:

```
\def\DEG#1{\relax \DEGa#1;''\relax}
\def\DEGa#1;#2'#3'#4\relax{%
  \evaldef\OUT{#1}%
  \ifnum\apSIGN<0 \def\tmps{-}%
  \else \def\tmps{+}\fi
  \DEGb#2;\relax{\OUT+\tmp/60}%
  \DEGb#3;\relax{\OUT+\tmp/3600}%
}
\def\DEGb#1;#2\relax#3{\edef\tmp{#1}%
  \ifx\tmp\empty \else
  \edef\tmp{\tmps\tmp}\evaldef\OUT{#3}\fi
}
```

The macro reads the argument using \DEGa, where #1 is the degrees, #2 minutes and #3 seconds. The first \evaldef calculates degrees. If the result is negative, we need to subtract the possible minutes and seconds (\tmps includes -).

The \DEGb macro removes the semicolon and next value. The raw value is stored in \tmp. Finally, \evaldef adds the minutes part and seconds part to the result \OUT.

Given the \DEG macro, we can now define macros \SINd, \COSd and \TANd. They accept the sexagesimal notation in its argument. For example, \SINd{12;30} is the sine of 12 degrees and 30 minutes.

```
\def\SINd#1{\relax \SIN{(\PI/180)*\DEG{#1}}}
\def\COSd#1{\relax \COS{(\PI/180)*\DEG{#1}}}
\def\TANd#1{\relax \TAN{(\PI/180)*\DEG{#1}}}
```

If a semicolon is not present in the argument, it is processed as a normal ⟨*expression*⟩ in degrees.

Maximum. We create the function-like macro

\MAX{⟨*expression*⟩,⟨*expression*⟩,...,⟨*expression*⟩}

Thus, the argument of this \MAX macro is a comma-separated list of any number of ⟨*expression*⟩s (at least one ⟨*expression*⟩ is needed). The macro returns the maximum of all given ⟨*expression*⟩s.

We cannot use the \ifnum or \ifdim primitives for comparison of two values because these values can be very large or have a very small difference, for example the first difference may be at the 50th decimal digit after decimal point. The documentation apnum.pdf recommends to use the \TEST macro, which can be defined:

```
\def\TEST#1#2#3#4{%
  \evaldef\tmp{#1-(#3)}\ifnum\apSIGN #2 0 }
```

and used:

\TEST {⟨*expression1*⟩} ⟨*relation*⟩ {⟨*expression2*⟩}
  \iftrue ⟨*true part*⟩ \else ⟨*false part*⟩ \fi

where ⟨*relation*⟩ is one of the characters <, >, =. The implementation of the \TEST macro is based on subtraction of the given two ⟨*expression*⟩s and testing the resulting \apSIGN. Note that the space after zero in the \TEST definition is needed because it closes the scanning of the zero number.

The \MAX implementation looks like this:

```
\newcount\mynum
\def\TEST#1#2#3#4{%
  \evaldef\tmp{#1-(#3)}\ifnum\apSIGN #2 0 }
\def\MAX#1{\relax \MAXa#1,,}
\def\MAXa#1,{%
  \evaldef\maxOUT{#1}\mynum=\apSIGN \MAXb}
\def\MAXb#1,{%
  \ifx,#1,\let\OUT=\maxOUT \apSIGN=\mynum \else
    \evaldef\maxNEXT{#1}%
    \ifnum\apSIGN>\mynum
       \mynum=\apSIGN \let\maxOUT=\maxNEXT
    \else \TEST\maxNEXT>\maxOUT \iftrue
       \let\maxOUT=\maxNEXT \fi\fi
    \expandafter \MAXb \fi
}
```

The \MAXa macro prepares the first value into \maxOUT and the sign of this value is \mynum. Then \MAXb is called repeatedly until #1 is empty. If #1 is empty, the resulting \apSIGN is set from \mynum and \OUT is \maxOUT; else, the next expression #1 is evaluated. If the sign of this partial result is

greater than the sign of the current result, then we do not need to execute the \TEST macro and can simply set new \maxOUT and \mynum. Else \TEST is processed and new value of \maxOUT is set only if the new result is greater than the current result.

We can create the analogous macro \MIN without copying all this code. We can just define the macro \mmREL as < or > and then use this macro instead of the > character in the above code.

Linear interpolation. We can use "declaration macros" \setF...\endF to define the function \F via a table of values. For example:

```
\setF
  \F{2} = 15 ;
  \F{3} = 10 ;
  \F{8} = 11 ;
\endF
```

Some \F{⟨*expression*⟩} = ⟨*expression*⟩; entries are listed here. A finite number of values of the function \F is given this way. Next, we define the function-macro \F{⟨*expression*⟩} which returns the value of the given \F using linear interpolation.

For the sake of simplicity, we don't implement a sorting algorithm and instead assume that the input values are sorted by the user. The previous example complies with this condition because $2 < 3 < 8$. Next, we suppose that the function \F is undefined outside the boundary input values (i. e. outside the $[2, 8]$ interval in our example). If the user tries to evaluate \F outside this interval then an "out of range" message is printed.

The \setF macro saves the information to the \Flist macro. This is a list of input values, i. e. 2;3;8; in our example. Moreover, the macros \F:⟨*number*⟩ are defined as the function value. The first entry has the number 1, second 2, etc. In our example, the \F:1 macro is defined as 15, \F:2 as 10 and \F:3 as 11. The \setF macro is defined by this code:

```
\newcount\mynum
\def\setF{\mynum=0 \def\Flist{}\setFa}
\def\endF{end setF}
\def\setFa#1{%
  \ifx#1\endF \else \expandafter \setFb \fi
}
\def\setFb#1#2#3;{%
  \evaldef\X{#1}%
  \evaldef\Y{#3}%
  \advance\mynum by1
  \expandafter
    \edef\csname F:\the\mynum\endcsname{\Y}%
  \edef\Flist{\Flist\X;}%
  \setFa}
```

The function-like macro \F evaluates the input parameter x = \X and scans the \Flist contents to find the sub-interval I of two consecutive input values where $x \in I$. The boundary values of the interval I are denoted by a = \A and b = \B, i. e. $I = [a, b)$. The values $F(a)$ = \FA, $F(b)$ = \FB are known and linear interpolation is applied:

$$\texttt{\OUT} = F(x) = F(a) + \frac{(x-a)\big(F(b)-F(a)\big)}{b-a}$$

```
\def\TEST#1#2#3#4{%
  \evaldef\tmp{#1-(#3)}\ifnum\apSIGN #2 0 }
\def\F#1{\relax
  \evaldef\X{#1}%
  \expandafter\Fa\Flist;\endF}
\def\Fa#1;{%
  \TEST{#1}>\X \iftrue
    \Fe \expandafter \Fc \fi % out of range
  \def\A{#1}\mynum=0
  \Fb
}
\def\Fb#1;{%
  \advance\mynum by1
  \ifx;#1;% the last item
    \TEST\X=\A \iftrue
      \evaldef\OUT{\Xa}\else \Fe \fi % out range
    \expandafter \Fc \fi
  \TEST{#1}>\X \iftrue
    \def\B{#1}%
    \edef\FA{\csname F:\the\mynum\endcsname}%
    \advance\mynum by1
    \edef\FB{\csname F:\the\mynum\endcsname}%
    \evaldef\OUT{\FA+(\X-\A)*(\FB-\FA)/(\B-\A)}%
    \expandafter \Fc
  \fi
  \def\A{#1}%
  \Fb
}
\def\Fc#1\endF{}
\def\Fe{\def\OUT{0}\apSIGN=0
  \message{F{\X} OUT OF RANGE}}
```

Printing and evaluating from shared source

So far, we have seen how the apnum package evaluates ⟨*expression*⟩s. As of version 1.5, apnum is able to print the same ⟨*expression*⟩s in math mode. The format of printing is determined automatically and is close to mathematical tradition. This is done with the \eprint{⟨*expression*⟩}{⟨*declaration*⟩} macro. You can specify the identifiers of "variables" in the ⟨*declaration*⟩.

We illustrate this feature by defining a macro \ep{⟨*expression*⟩}. It prints the ⟨*expression*⟩ and then evaluates the same ⟨*expression*⟩, showing the

value. The macros \X, \Y and \Z are variables for these examples.

```
\def\vars{\def\X{x}\def\Y{y}\def\Z{z}}
%
\def\ep#1{$\displaystyle
  \eprint{#1}\vars% printing
  \evaldef\OUT{#1}% evaluation
  \ROUND\OUT6%      round result to 6 digits
  \corrnum\OUT    % .digits -> 0.digits
  \ifx\XOUT\empty =\else\doteq\fi \OUT$}
```

We need to give values to the variables x, y, z before starting the experiment:

```
\def\X{0.51} \def\Y{-2.7}  \def\Z{17}
```

Now, let's apply the \ep macro in a variety of cases:

```
\ep{(\X^2+1)/((\X+1)*(\X-2))}
```

$$\frac{x^2+1}{(x+1)\cdot(x-2)} \doteq -0.560069$$

```
\ep{-((\X^2-1)/((\X+1)*(\X-1)))}
```

$$-\frac{x^2-1}{(x+1)\cdot(x-1)} = -1$$

```
\ep{\SIN{\Y}^2 + \COS{\Y}^2}
```

$$\sin^2 y + \cos^2 y \doteq 0.999999$$

```
\ep{\ASIN{\X} + \ATAN{\X+1}}
```

$$\arcsin x + \arctan(x+1) \doteq 1.521041$$

```
\ep{\SIN{\PI/4}}
```

$$\sin\frac{\pi}{4} \doteq 0.707106$$

```
\ep{\SQRT{2}/2}
```

$$\frac{\sqrt{2}}{2} \doteq 0.707106$$

```
\ep{\PI}
```

$$\pi \doteq 3.141592$$

```
\ep{\FAC{\Z}}
```

$$z\,! = 355687428096000$$

```
\ep{\SQRT{\iFLOOR{\Y}^2+1}}
```

$$\sqrt{\lfloor y\rfloor^2+1} \doteq 3.162277$$

```
\ep{\iFLOOR{\Y} + \iFRAC{\Y}}
```

$$\lfloor y\rfloor + \{y\} = -2.7$$

```
\ep{\LN{\X/\Y^2}+1}
```

$$\ln\frac{x}{y^2}+1 \doteq -1.659848$$

```
\ep{(\X+\Y)*-3}
```

$$(x+y)\cdot(-3) = 6.57$$

```
\ep{-3*-(\X+\Y)}
```

$$-3\cdot(-(x+y)) = -6.57$$

```
\ep{\BINOM{5}{1}+\BINOM{5}{2}}
```

$$\binom{5}{1}+\binom{5}{2} = 15$$

```
\ep{2^5/2}
```

$$\frac{2^5}{2} = 16$$

```
\ep{4^3^2}
```

$$4^{3^2} = 262144$$

```
\ep{(4^3)^2}
```

$$\left(4^3\right)^2 = 4096$$

```
\ep{\EXP{\LN{2}+\LN{3}}}
```

$$\mathrm{e}^{\ln 2+\ln 3} \doteq 5.999999$$

Note that the \eprint macro does not insert redundant parentheses and follows traditional math typesetting. For example \SIN{\X}^2 prints as $\sin^2 x$. On the other hand, new parentheses are sometimes needed, for example -3*-(\X+\Y) is printed in the form $-3\cdot(-(x+y))$.

About the implementation

The algorithms used are described in detail in the technical part of the apnum.pdf documentation. This section introduces only the basic ideas.

Expression interpreter. When I was young (about 15 years old) I was a participant in a hobby course on programming. We were working with a mainframe EC 1010. Our teacher taught me how to program an expression interpreter (with operators of various priorities) using stacks. My first implementation of this was in FORTRAN. Now, many years later, I was able to use this knowledge and implemented the expression scanner again, now in apnum. The apnum package implements the expression scanner in two steps: first the ⟨*expression*⟩ is converted to Polish notation and this format is used for evaluating (or printing) in the second step.

Basic operations. Addition, subtraction, multiplication and division are implemented similarly to the way pupils learn to do these operations in school. The main difference is the base of the number system used. Students use base 10, manipulating with the ten different digits of this system and drilling the "small multiplication table" up to 100. On the other hand, apnum uses a number system with base 10000, each "digit" has up to four decimal digits and TEX

supports a "multiplication table" to 10^8 using the `\multiply` primitive. This is possible because the maximum number that can be represented in TEX registers is $2^{31} \doteq 2 \cdot 10^9$. For example, to multiply two ten digit numbers, pupils need to do 100 multiplications but TEX needs only 9 multiplications.

TEX can only do direct access to its memory using new macro definitions. But this is not a good approach for implementing "digits" values. I did many tests of various methods. I found that the linear access to the sequence of "digits" in the input stream is the most efficient. Data are expanded to the input stream and read again. One problem is that we have only one input stream, but we typically need to read digits from two sources. So `apnum` uses a special interleaved format for these calculations. The data are converted from human-readable form to this interleaved format when we need to convert pairs of four decimal digits to one internal "digit". Then the calculation is processed (typically in a loop) over this interleaved format.

The division algorithm is well known from school too: the "tail" of partially calculated remainders is constructed. The `apnum` package optimizes this processing if the divisor is only one "digit" (i. e. at most four decimal digits). Then the complexity of division depends linearly on the desired number of digits in the result. When the divisor has more "digits", then `apnum` uses the special interleaved data format mentioned above.

Many other optimizations were done. For example, suppose a big number with many digits is given in the parameter `#1` and a macro is written roughly like this:

```
\def\macro#1{%
  \ifA \ifB  do something{#1}%
       \else do something{#1}\fi
  \else do something{#1}\fi}
```

This `\macro` approach above is not a good idea. Why? Because the big parameter is expanded three times here, thus much data is skipped many times by `\if...\else...\fi` primitives. This is time-consuming. So it is much better to do `\def\tmp{#1}` at the beginning of `\macro` and then do skipping over `\tmp` only.

Mathematical functions. As a student I did a school assignment on "long numbers" on the mainframe installed at our university. Punch cards were used. I implemented addition, subtraction, multiplication and division. My dream was to continue with this work and implement classical math functions as well. But lack of time and the unsuitable technology was too much of barrier, so the dream wasn't realized at that time. But now, I returned to my student days and started to implement math functions in `apnum`. The dream has been fulfilled now.

Square root. One of the algorithms for computing square roots is similar to the division algorithm, but its complexity isn't linear with the number of desired digits in the result. I had started working on this algorithm on the mainframe as a student, but now, I decided to use Newton's method.

We need to find the first approximation x_0 of $\sqrt{a}$. Then the tangent line to the graph of the function $f(x) = x^2 - a$ is constructed in the point $[x_0, f(x_0)]$. The position of the tangent can be found using calculus. The intersection of this line with the x axis is the next approximation of $\sqrt{a}$. This step is repeated until the desired precision is reached. I decided to use the linear interpolation of the function $\sqrt{x}$ in the interval $[1, 100]$ for calculating the first approximation used by Newton's method. The linear interpolation uses known values in the points $1, 4, 9, 16, \ldots, 81, 100$. Only classical TEX operations (no `apnum` operations) are used for calculating the first approximation. If we need 20 digits in the result then 5 iterations of Newton's method is sufficient because the number of calculated digits is doubled in each iteration step and the linear interpolation starts with 1 digit calculated.

If the argument is outside the interval $[1, 100]$ then we can shift its decimal point by an even number M of positions. Then we do the calculation of square root. Finally, we shift the decimal point back by $M/2$ positions in the result. This idea is based on the fact that $\sqrt{100} = 10$.

Exponential. The well-known Taylor series is used in `apnum`:

$$e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots$$

This series converges well for $|x| < 1$ because of the factorial in the denominators. But what to do if the argument x is outside $[-1, 1]$? First of all, negative arguments are converted to positive using identity $e^{-x} = 1/e^x$. If the argument $x \geq 4$ then we calculate $d = \lfloor x / \ln 10 \rfloor$ and use the identity

$$e^x = e^{x - d \cdot \ln 10} \cdot 10^d$$

This means that we need to calculate the exponential of the argument $x' \in [0, \ln 10] \subset [0, 4)$ and then we shift the decimal point of the result by d digits.

If the argument $x \in [1, 4)$ then we divide it by two or by four in order to have $x' \in [0, 1)$. Then we use the Taylor series mentioned above for x' and

finally the result is $(e^{x'})^2$ if $x' = x/2$ or $((e^{x'})^2)^2$ if $x' = x/4$. This is based on the identity $e^{2x} = (e^x)^2$.

We need to do about 20 steps in the Taylor series for 20 digits of precision because $20! \approx 10^{19}$.

Logarithm. The following series derived from the inverse of the hyperbolic tangent is used:

$$\ln x = 2 \tanh^{-1} \frac{x-1}{x+1} =$$
$$= 2\left(\frac{x-1}{x+1} + \frac{1}{3}\left(\frac{x-1}{x+1}\right)^3 + \frac{1}{5}\left(\frac{x-1}{x+1}\right)^5 + \cdots\right).$$

The disadvantage of this series is that it converges well only for $x \approx 1$. But we are able to modify the argument so that it is approximately equal to one. First, we calculate $A = x/\exp(\widetilde{\ln x})$, where $\widetilde{\ln x}$ is an approximation of $\ln x$. We use linear interpolation. It is evident that $A \approx 1$ because $\exp(\ln x) = x$, if exact $\ln x$ is used. Next, we calculate $\ln A$ using the series above. Finally, $\ln x = \ln A + \widetilde{\ln x}$ because $x = A \cdot \exp(\widetilde{\ln x})$ and $\ln(ab) = \ln a + \ln b$.

This algorithm need be implemented only for $x \in [1, 10)$. If the argument is outside this interval, then we shift the decimal point by M positions and then calculate $x = x' \cdot 10^M$, $\ln x = \ln x' + M \cdot \ln 10$. The frequently used value $\ln 10$ is calculated to the needed precision only once and saved into memory.

Because the linear interpolation for $\widetilde{\ln x}$ differs from the exact result at the second decimal digit, the same is true for the difference between A and 1. Each step of the Taylor series improves precision by four digits because there are only odd powers.

Sine and cosine. The known Taylor series for sine and cosine are similar to the Taylor series for the exponential. So, we need to have the argument in the interval $[0, 1)$. We can shift it by a multiple of period (or half-period), but we need to know the constant π first. The `apnum` package calculates and saves π to 30 digits in its memory. If more precision is desired then π is re-calculated by the Chudnovsky formula. It converges very well, with 14 new exact digits per one step. It has only one problem: to calculate $\sqrt{640320}$. This constant is used in the Chudnovsky formula. So `apnum` stores the initial approximation for Newton's method (for $\sqrt{x}$ calculation) with 12 decimal digits for this special case. This saves several steps of Newton's method.

After the sine or cosine argument x is shifted by a half-period multiple, we have $x \in [0, \pi)$. If x is outside of $[0, \pi/2)$ then we can use the identities $\sin x = \sin(\pi - x)$ or $\cos x = -\cos(\pi - x)$. Now, we have a new argument $x \in [0, \pi/2)$. If x is outside the interval $[0, \pi/4)$ then we can use identities $\cos x = \sin(\pi/2 - x)$ or $\sin x = \cos(\pi/2 - x)$. The new argument is in the interval $[0, \pi/4) \subset [0, 1)$ and the Taylor series for sine or cosine can be used.

Inverse of tangent. The function $\tan^{-1} x = \arctan x$ is implemented by the series for the argument $1/x$:

$$\arctan \frac{1}{x} = \frac{x}{1+x^2} + \frac{2}{3}\frac{x}{(1+x^2)^2} +$$
$$+ \frac{2}{3}\frac{4}{5}\frac{x}{(1+x^2)^3} + \frac{2}{3}\frac{4}{5}\frac{6}{7}\frac{x}{(1+x^2)^4} + \cdots$$

It converges well for $x > 1$. If $x \in (0, 1)$ then we can use the identity $\arctan x = \pi/2 - \arctan 1/x$ and if the argument is negative we use the fact that the function is odd.

Other common mathematical functions can be expressed directly with the functions mentioned above.

The final joke

The `apnum` package uses only TEX primitives and the basic plain TEX macro `\newcount`. Thus, the package works in classical or extended TEX with any format. This is the general approach of almost all my macros. On the other hand, typical LATEX packages require the LATEX format and don't work with anything else. This is shown explicitly for example by `\NeedsTeXFormat{LaTeX2e}` in such macros. The LATEX macros are usually a mix of TEX primitives and LATEX constructs: a mix of `\def` and `\newcommand`, a mix of `\newcount` and `\newcounter`, a mix of `\advance` and `\addtocounter`, a mix of `\hbox` and `\mbox`, a mix of `\setbox` and `\sbox`, a mix of `\vrule` and `\rule` etc. Pure TEX macros (which can be used in plain TEX too) are infrequent in the LATEX world, unfortunately (in my view).

So, I decided to put the following code at the end of my `apnum.tex`:

```
% please, don't remove this message
\ifx\documentclass\undefined \else
\message{WARNING: the author of apnum
  package recommends: Never use LaTeX.}\fi
```

Thus the above message is printed on the terminal and in the log file when LATEX is used. This expresses my opinion about LATEX. And I hope that this does not matter, because a typical LATEX user reads neither the log file nor the terminal output, because plenty of useless information is printed there.

◇ Petr Olšák
Czech Technical University in Prague
http://petr.olsak.net