A really fast method for converting an arbitrary precision number to a decimal string.

By Henrik Vestermark (hve@hvks.com)

Abstract:

We all know how to conversion of simple language types like integers and float to a decimal string using basic encoding technics. However, how does it stack up when using it for an arbitrary precision variable with thousands, or even millions of digits to convert to Decimal string form?
This paper describes the issue with the basic conversion algorithm and outlines a new improved algorithm that speeds up the process by a factor of approx. 100,000.

Introduction:

This paper come to light when I was finished writing a new version of my Arbitrary precision library after I converted the internal format from a decimal base string form (where each arbitrary precision number was stored as a decimal string) to a binary form (where the arbitrary precision number was stored as a vector of binary digits). In the conversion, processed everything when smoothly until I began to test the performance of the library. The internal arithmetic handling was fast for all the usual types like +,-,*,/ etc as expected when using binary digits instead of decimal digits. However when I was testing the initialization of an arbitrary precision number with a string and the conversion back from the internal binary form to output the decimal representation of the number, the performance when 'south' when handling more than a few thousand digits. To improve the performance I outline a new way to handle this conversion for large numbers of a million digits or more.

Change log

2-March-2023. Cleaning up the document and making minor changes.

Contents

A simple approach for conversion to string form

As taught in computer language classes, we learn how to output a decimal string of binary numbers and there are several possibilities to do this.

1) Use the library sprint() function.
2) Use itoa() if you were dealing with integers.
3) Use cout << variable;
4) Using stringstream.
5) Write your little code block.
6) Many others methods.

To make it more general we will initially use standard floating-point numbers (float or double) to output a string, based on the STL library string class. A subset of the conversion deals with the integer portion of the float number which is also covered. Usually in the process, we will do one decimal digit conversion at a time and the following pseudo algorithm can be used. The algorithm is based on splitting the float number into an integer part and a fraction part and luckily, we do have a standard C library function that does just that. This is the modf() function calls see e.g. modf - C++ Reference (cplusplus.com)

The algorithm is as follows:

1) Split the integer part (ip) and fraction part (fp) part from float number f.
   fp=modf(f, &ip);
2) Convert the Integer part first.
   str=integer2digits(ip);
3) Convert the Fraction part next.
   str+=”.”+fraction2digits(fp);

The two functions integer2digits() and fraction2digits() can easily be written as follows together with the main function float2digits()

```
// Convert an integer to a decimal string
string integer2digits(float_precision& ip)
       {
       string str;
       float_precision fp;
       while (ip.iszero()==false)
              {
              fp = modf(ip/10.0, &ip);
              fp *= 10.0;
              str += (int)round(fp) + '0';
              }
       // We collected the digits in backward order so we reverse the string
       reverse(str.begin(), str.end());
       return str;
       }

// Convert a fraction to a decimal string
```

```
string fraction2digits(double fp)
       {
       string str;
       double ip;
       // Len is the number of decimal digits in a double
       size_t len = (int)(53 / log2(10));
       while (fp != 0.0 && str.length() < len)
              {
              fp *= 10.0;
              fp = modf(fp, &ip);
              str += (int)ip + '0';
              }
       return str;
       }

// Convert float to a decimal string
string float2digits(double f)
       {
       string str;
       double fp, ip;

       fp = modf(f, &ip);
       str = integer2digits(ip);
       str += "." + fraction2digits(fp);
       return str;
       }
```

All in all, pretty much basic stuff.

The Arbitrary precision library

If you already are familiar with the arbitrary precision library, you can skip this section.

To understand the C++ code and text we have to highlight a few features of the arbitrary precision library where the class name is *float_precision*. Instead of declaring, a variable with a float or double you just replace the type name with *float_precision*. E.g.

```
float_precision f;  // Declare an arbitrary precision float with 20 decimal digits precision
```

You can add a few parameters to the declaration. The first is the optional initial value and the second optional parameter is the floating-point precision. The native type of a *float* has a fixed size of 4 bytes and 8 bytes for *double*, however since this precision can be arbitrary we can declare the wanted precision as the number of ***decimal digits*** we want to use when dealing with the variable. E.g.

```
float_precision fp(4.5);  // Initialize it to 4.5 with default 20 digits precision
float_precision fp(6.5,10000); // Initialize it to 6.5 with a precision of 10,000 digits
```

The precision of a variable can be dynamic and change throughout the code, which is very handy to manipulate the variable. To change or set the precision you can call the method .precision() E.g.

```
f.precision(100000);              // Change the precision to 100,000 digits
f.precision(fp.precision()-10);  // Lower the precision with 10 digits
f.precision(fp.precision()+20); // Increase precision with 20 digits
```

There is another method to manipulate the exponent of the variables. The method is called .exponent() and returns or sets the exponent as a power of two exponents (same as for our regular build-in types *float* and *double*) E.g.

```
f.exponent();            // Return the exponent as 2^e
f.exponent(0)            // Remove the exponent
f.exponen(16)            // Set the exponent to 2^16
```

There is a second way to manipulate the exponent and that is the class method. .adjustExponent(). This method just adds the parameter to the internal variable that holds the exponent of the float_precision variable. E.g.

```
f.adjustExponent(+1); // Add 1 to the exponent, the same as multiplying the number with 2.
f.adjustExponent(-1);  // Subtract 1 from the exponent, the same as dividing the number with 2.
```

This allows very fast multiplication of division with a number that is any power of two.

The method .iszero() returns true if the float_precision number is zero otherwise false. There is an additional method() but I will refer to the reference for the user manual to the arbitrary precision math package for details.

All the normal operators and library calls that work with the built-in type float or double will also work with the float_precision type using the same name and calling parameters.

Internal format for float_precision variables

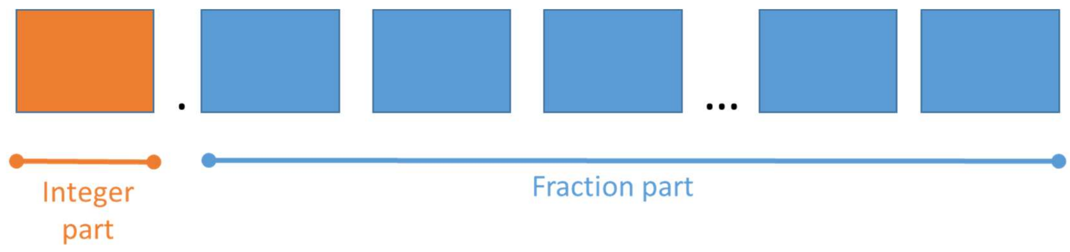
For the internal layout of the arbitrary precision number, we are using the STL vector library declared as:

```
vector<uintmax_t> mBinary;
```

uintmax_t is mostly a 64-bit quantity on most systems, so we use a vector of 64-bit unsigned integers to store our floating-point precision number.

The method .size() returns the number of internal vector entries needed to hold the number.

The Binary format mBinary



- The binary format consist of an unlimited number of 64bit unsigned integer blocks.
- One block in front of the period sign '.' (the integer part of the number)
- Zero or more blocks of fractions after the '.' (the fraction sign of the number)
- The binary number is stored in a STL vector class and defined
    - vector<uintmax_t> mBinary;
- There is always one entry in the mBinary vector.
- Size of vector is always >=1
- A Number is always stored normalized. E.g. the integer part is 1 or zero
- The sign, exponent, precision, rounding mode is stored in separate class fields.

There are other internal class variables like the sign, exponent, precision, and rounding mode but these are not important to understand the code segments.

Normalized numbers

A float_precision variable is always stored as a normalized number with a one in the integer portion of the number. The only exception is zero, which is stored as zero. Furthermore, a normalized number has no trailing zeros.

For more details see [1].

Arbitrary precision conversion to string form

With the float_precision class explained we can easily transform our three functions into an arbitrary precision form. See below. Notice there are a few extra calls to the method precision() to accommodate the dynamic nature of an arbitrary precision number.

```
// Convert an integer to a decimal string
string integer2digits(float_precision& ip)
	{
	string str;
	float_precision fp;
	fp.precision(ip.precision());
	for (; ip.iszero()==false;)
		{
		fp = modf(ip / 10.0, &ip);
		fp *= 10.0;
		str += (int)round(fp) + '0';
```

```
            }
        // We collected the digits in backward order so we reverse the string
        reverse(str.begin(), str.end());
        return str;
        }

// Convert a fraction to a decimal string
string fraction2digits(float_precision& fp)
        {
        string str;
        float_precision ip;
        // Len is the number of decimal digits in a double
        size_t len = fp.precision();
        ip.precision(len);
        for (; fp.iszero() &&str.length() < len;)
                {
                fp *= 10.0;
                fp = modf(fp, &ip);
                str += (int)ip + '0';
                }
        return str;
        }

// Convert float to a decimal string
string float2digits(float_precision&f )
        {
        string str;
        float_precision fp, ip;

        fp.precision(f.precision());
        ip.precision(f.precision());
        fp = modf(f, &ip);
        str = integer2digits(ip);
        str += "." + fraction2digits(fp);
        return str;
        }
```

While performance is not too bad for the built-in type of float and double, the arbitrary precision performance is terrible. See the performance table below.

Digits	Time (sec)
10,000	5
100,000	987
1,000,000	363,373

The reason is that the statement fp*=10.0; in both functions becomes increasingly expensive to execute when precision increases and then we are only getting one decimal digit per loop out of our effort.

Bottom line is that the above algorithm is not sufficient for dealing with arbitrary precision variables.

2nd approach

Since the performance issue is the multiplication of fp*=10.0 and a single digit per loop approach we developed the idea of handling more than one digit at a time. Each of our internal vectors is a 64-bit unsigned integer quantity with a range from 0…18446744073709551615 and a 64-bit integer can therefore hold 19 decimal digits for sure, so why not do 19 decimal digits at a time instead of 1. We would need an extra function delivering the 19 digits and in case we do not have 19 digits, it will be padded in front with ‘0’.

The function uitostring10() below delivered up to 19 digits at a time in a string form.

```
string uitostring10(const uintmax_t value, const unsigned minlength = 0)
       {
       std::string s;
       unsigned digit;
       uintmax_t uvalue = value;

       do
              {
              digit = (unsigned)(uvalue % 10);
              uvalue /= 10;
              s.push_back(digit+'0');
       } while (uvalue > 0);

       while (s.length() < minlength)
              s.push_back('0');
       reverse(s.begin(), s.end());
       return s;
       }
```

And we now multiply in the function fraction2digits() with:
fp*=10e19 instead of fp*=10
and replace:
`str += (int)ip + '0';` with `str += uitostring10((int)ip,19);`

```
// Convert a fraction to a decimal string
string fraction2digits(float_precision& fp)
       {
       string str;
       float_precision ip;
       // Len is the number of decimal digits in a double
       size_t len = fp.precision();
       ip.precision(len);
       for (; fp.iszero() && str.length() < len;)
              {
              fp *= 10E19;
              fp = modf(fp, &ip);
              str += uitostring10((int)ip,min(19,(int)(len-str.length())));
              fp.precision(fp.precision()-19);
              }
       return str;
       }
```

Of course, when we are down to the last iteration we could need to handle between 1..19 digits and we would need to multiply with less than 1E19.

There is one more trick we apply. Since we are getting 19 decimal digits at a time, we can reduce the precision by 19 digits for each iteration using the statement:

```
        fp.precision(fp.precision()-19);
```

This small trick doubles the performance of the loop, so it is worth applying.

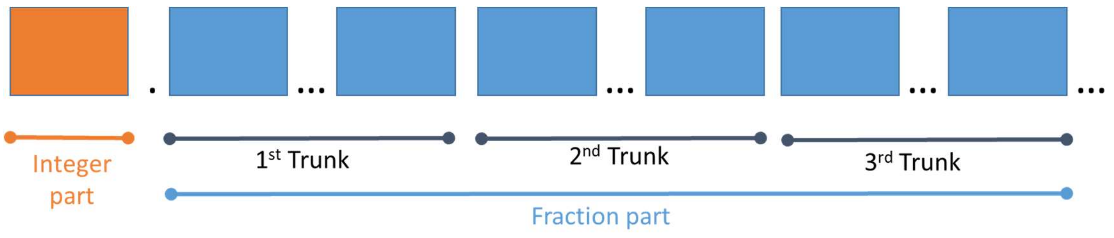
Our 2nd approach test result is below.

Digits	Time (sec)	
	1st Approach	2nd Approach
10,000	5	0.3
100,000	987	52
1,000,000	363,373	9,270

A significant improvement. The gain is approximately a factor of 39 (1M digits) following our expectation when we do 19 digits at a time and use precision reduction that generates a gain of 2. The total expected gain of 19·2=38 is very close to our measured gain of 39. However, when numbers exceed 1,000 digits we see a slowdown that also makes our second approach insufficient for our needs.

3rd Approach

In a continuation of the second approach, we can instead extract a smaller portion of the fraction part, work on that portion, then adjust the fraction part, and then repeat it. We will define a trunk as several binary elements in the mBinary vector.
Let's say 50 binary indexes in the vector is a trunk. (The trunk size of 50 is found by experience and testing). The first trunk is the binary vector mBinary[1..50]. The second trunk is mBinary[51.100] etc.



The idea is to work on only a portion of the entire fraction part at a time. Now for each of the 50 elements in a trunk we can, for sure get 19 decimal digits. For the entire trunk, we can get 19·50=950 decimal digits that we can process quickly since we are dealing with a much smaller number of digits at a time. Since we extract 50x64bit quantities we take out

3,200 bits enough to hold a least ~ 963 digits (3,200/log2(10)). That gives us a sufficient margin for our goal of extracting 950 digits. After processing the trunk we then multiply the ***original*** fraction part with 10^{950}, discard the integer part and repeat the process for the new remaining fraction part. E.g.

```
        fp=modf(fp*10E950,&ip);
```

We have implemented this as a function *trunkfraction2digits()* that converts a single trunk into decimal digits and returns the decimal string. We also notice that we can't just multiply the fraction with a constant as large as 10E950 since it will exceed the limits of a double. We need to create a new floating_point precision variable called *trunkPowerof950* that is initialized through a string "10E950" which has no upper limit.

```
// Convert a single trunk of 950digits to decimal
string trunkfraction2digits(float_precision& fp)
       {
       std::string str;
       float_precision trunk(fp),ip;
       // A trunk is 50 * 19 digits or 950digits
       static float_precision trunkPower950("10E950", 1000);

       trunk.precision(950);
       str.reserve(950);
       // Do it trunk size of 19 digits
       str += fraction2digits(trunk);
       fp = modf(fp*trunkPower950, &ip);
       // Lower precision with a trunk size
       fp.precision(fp.precision() - 950);
       return str;
       }
```

And the main function *float2digits()* change to:

```
// Convert float_precision to decimal string
string float2digits(float_precision&f )
       {
       string str;
       float_precision fp, ip;

       fp.precision(f.precision());
       ip.precision(f.precision());
       fp = modf(f, &ip);
       str = integer2digits(ip);
       str += ".";
       while (fp.precision() >= 950)
              str += trunkfraction2digits(fp);
       str += fraction2digits(fp);
       return str;
       }
```

Again, we see a significant improvement in performance

Digits	Time (sec)		
	1st Approach	2nd Approach	3rd Approach

10,000	5	0.3	0.011
100,000	987	52	1.1
1,000,000	363,373	9,270	183

And the gain is in the range of a factor of ~ 50 from the 2nd approach.

4th Approach

By further grouping, the trunks into new groups of 10 outer trunks we can further divide the workload into groups of smaller trunks called kilo trunks and we again see a significant performance improvement. A kilo trunk can at least contain 10·50·19=9,500 digits.

Digits	**Time (sec)**			
	1st Approach	2nd Approach	3rd Approach	4th Approach
10,000	5	0.3	0.011	0.013
100,000	987	52	1.1	0.23
1,000,000	363,373	9,270	183	19.4

And the gain is in the range of a factor of ~ 10 from the previous approach.

5th Approach

By further grouping the kilo trunks into groups of 10-kilo trunks, we get our mega trunk. A mega trunk contains 10·10·50·19=95,000 digits.

Digits	**Time (sec)**				
	1st Approach	2nd Approach	3rd Approach	4th Approach	5th Approach
10,000	5	0.3	0.011	0.013	0.014
100,000	987	52	1.1	0.23	0.27
1,000,000	363,373	9,270	183	19.4	4.3

Now we have an acceptable result for up to approx. 1M digits. You can continue this kind of grouping for better performance for numbers exceeding 1M digits.

In the arbitrary precision packages which can be downloaded from Arbitrary Precision C++ Packages, we use two extra trunk levels working on 950,000 and 9,500,000 digits respectively at a time.

Arbitrary precision for very small or big numbers

Until now we have only dealt with numbers with very high precision, however, to get the full picture we also need to see what happens with very small and very large numbers. What we are thinking about is numbers with a very high negative exponent. E.g. 1.23E-

^{10000} or very high positive exponent e.g. 1.23E$^{+10000.}$ Using the algorithm from the previous section, we get a very high number of leading or trailing zeros:

1.23E^{-10000} = $0.\underline{000000\ldots0000}123$ where the underscores zero is equivalent with 10,000 zeros or 1.23+10000 = $123\underline{00000\ldots0000}$ and again where the underscored zero is equivalent with 9998 zeros.

Of course, it is not optimal to generate that amount of (zeros) number just to discard them again when displaying the number in exponential notation.

As mentioned in the arbitrary precision numbers, the exponent value is stored as a signed integer in base two just as the float or double variable in C or C++.

Arbitrary precision for very small numbers

Internally we have our exponent in base two and want it to be displayed in exponential notation in base 10. The first question to answer is what is an exponent in base 2 worth in base 10. The answer is $\text{exponent}_{\text{base2}}/\log2(10)$.Now we can't deal with this as a float value but need to find the integer in base 10 exponent that ensures:

$$\text{exponent}_{\text{base2}} >= \text{exponent}_{\text{base10}}$$

By truncating, the value to an integer we ensure that the above requirement is handled.

Example. Let us assume we have an arbitrary precision float number $1*2^{-20000}$ Since the internal exponent is in base 2. By dividing -20000 with log2(10) you get -6020.5999 which truncated to an integer gives you a decimal exponent in base 10 at -6020. So instead of 6020 zero after the '.' sign you instead have found the exponent in base 10 as E-6020. And we can then eliminate the first 6020 zeros by multiplying the number we need to convert to decimal representation by a factor of 10^{6020} We know that the remaining number will start with a non-zero value and we can use the algorithm from the previous section to get the remaining decimal up to the precision of the float number.

Arbitrary precision for very big numbers

Conclusion

The above methods show that you can gain a significant performance improvement by slicing and further subdividing the job into smaller trunks for processing. Going from 363,373sec to 4.3s is an improvement of a factor of more than 84,000 times for processing 1M digits to decimal form.

Reference

1) Arbitrary precision library package. Arbitrary Precision C++ Packages (hvks.com)

Appendix

Below is the actual function in the Author's Arbitrary Precision Maths library that converts the internal format to a decimal string form. As can be seen, there is a little more to the story than just explained in this paper.

```
//	@author Henrik Vestermark (hve@hvks.com)
//	@date		2/Dec/2021
//	@brief		Convert float_precision numbers into a string (decimal representation)
//	@return		std::string -	The decimal floating-point string
//	@param		"a"		float_precision number to convert
//
//	@todo
//
// Description:
//   Convert float_precision numbers into a string (decimal representation)
//   We do it in multiple steps.
//	1)	Repeat doing a trunk size by extracting a trunk size from the fptoa number
//		in the same manner as step 2
//	2) Repeat doing a trunk size by extracting a trunk size from fptoa number
//		Then repeat multiple groups within the trunk size of max_digis number at the time
//		The most efficient trunk size is 50*max_digits decimal in a trunk size
//		For every loop we reduce the original fptoa precision  with the trunk size precision
//	 3)	Take the remaining fpto number in the range 0..max_digits
//
std::string _float_precision_fptoa(const float_precision *a)
	{
	const size_t thr = MAX_TRUNK_SIZE*MAX_DECIMAL_DIGITS;
	const size_t kthr = thr * MAX_KILOTRUNK_SIZE;
	const size_t mthr = kthr * MAX_MEGATRUNK_SIZE;
	const size_t m10thr = mthr * MAX_MEGA10TRUNK_SIZE;
	const size_t m100thr = m10thr * MAX_MEGA100TRUNK_SIZE;
	const size_t gigathr = m100thr * MAX_GIGATRUNK_SIZE;
	float_precision fracp, intp;
	std::string str;
	int expo10, sign = 1;
	size_t found, len;

	if (a->iszero() )
		return std::string("0E0");
	sign = a->sign();
	fracp.precision(a->precision());		// ensure fracp and intp has the same precision as a
	intp.precision(a->precision());
	fracp = modf(*a, &intp);			// Separate the integer and fraction
	intp = fabs(intp);
	str += _float_precision_fptoainteger(&intp);	// Convert integer part to string
	expo10 = (int)(str.size() - 1);			// Extract Expo as the size of string - 1
	if (!fracp.iszero() || str.size() > 1)		// If fraction part then add "." to string
		str.insert(str.begin() + 1, '.');
	fracp = fabs(fracp);				// Remove any fraction sign if any
	fracp.precision(fracp.precision() + 2 );	// Add extra guard bits
	fracp.mode(ROUND_DOWN);
	// Check for a large negative exponent to avoid generating a leading zero that will be cut off anyway
at the end
	expo10 -= exponent2reduction(fracp);  // remove large negative exponent from fracp and adjust fracp
accordingly
	str.reserve(fracp.precision() + str.length() + 32);  // Ensure enough room,for the string to avoid
reallocating
	len = fracp.precision() -2 + str.length();// The expected length of the fraction before we have enough

	// Do it in Giga trunks size
	for (; (str.length() + gigathr < len) && !fracp.iszero(); )
		str += gigatrunk2Decimal(fracp);

	// Do it in 100M trunks size
	for (; (str.length() + m100thr < len) && !fracp.iszero(); )
		str += mega100trunk2Decimal(fracp);

	// Do it in 10M trunks size
	for (; (str.length() + m10thr < len) && !fracp.iszero(); )
		str += mega10trunk2Decimal(fracp);

	// Do it in mega trunks size
```

```
        for (; (str.length() + mthr < len) && !fracp.iszero(); )
                str += megatrunk2Decimal(fracp);

        // Do it in kilo trunks size
        for (; (str.length() + kthr < len) && !fracp.iszero(); )
                str += kilotrunk2Decimal(fracp);

        // Do it in trunks of threshold times (64bit numbers). threshold*max_digits
        for (; (str.length() + thr < len) && !fracp.iszero(); )
                str += trunk2Decimal(fracp);

        // Do it for the remaining less than a trunk size of the threshold
        for (; (str.length() < len || abs(fracp.exponent()) > 3) && !fracp.iszero(); )
                str += number2Decimal(fracp, (int)(len - str.length()));

        // Remove trailing zeros if any
        found = str.find_last_not_of('0');
        if (found != std::string::npos && str.size() > std::max(2, (int)found + 1))
                {
                str.erase(std::max(2, (int)found + 1));
                if (str[str.size() - 1] == '.')
                        str.erase(1);
                }

        // check and find a negative exponent.
        if (str[0] == '0')
                {
                found = str.find_first_not_of('0', 2);
                if (found == std::string::npos)
                        {// Everything is zero
                        return std::string("0E0");
                        }
                else
                        {// Reduce exponent with the number of leading zeros found
                        expo10 = -(int)(found - 1);
                        str.erase(0, found);
                        if (str.size()>1)
                                str.insert(str.begin() + 1, '.');
                        }
                }
        // str is on the form x.yyyyyyyyyyy
        // Remove any digits exceeding the precision
        if (str.size() > a->precision() + 2)
                str.erase(a->precision() + 2);

        // Add E[+-]exponent to the back of str and sign if negative to the front
        str = (sign < 0 ? "-" : "") + str+"E"+ itostring(expo10, BASE_10);
        return str;
        }
```