Practical implementation of Spigot Algorithms for Transcendental Constants

By Henrik Vestermark (hve@hvks.com)

Abstract:

This paper examined the various modern version of spigot algorithm for calculating transcendental constant like π, *e*, ln(2) and ln(10) to unlimited precision. It layout the algorithm and the timing for the constants and compare it with a traditional implementation using arbitrary precision arithmetic. It is found that the performance of spigot algorithm beats the traditional method using arbitrary precision with several factors and it is therefore recommend to be used instead, when performance is needed.

Introduction:

In a previous paper finding practical algorithms for π, I introduced the bounded spigot algorithm for finding π with arbitrary precision. This paper expand on this to also show that spigot algorithm can be useful for calculating other transcendental constant like *e*, ln(2) and ln(10)

The paper is divided into two sections. Section 1 is calculating transcendental *e* constants using bounded spigot algorithm while section 2 is dedicated to the unbounded versions. The bounded spigot algorithm is an alternative way of generating transcendental constants and does not require us to resort to arbitrary precision arithmetic but can stick with simple integer arithmetic in either 32-bit or 64-bit versions. As always, we list C++ source code for the practical implementation of theses algorithms.

Change log

This revision add the unbounded spigot algorithm for pi and make minor change throughout the document from the original paper from 2017.

Contents

BBP Notation

There exist a large number of series that can all be generalized for short hand using the following notation:

$$P(s,b,n,A)=\sum_{k=0}^{\infty}\frac{1}{b^k}\sum_{j=1}^{n}\frac{a_j}{(kn+j)^s}$$

Where s, b, n, are integers and A denote a vectors of integers $A=(a_1,a_2,\dots,a_n)$.

Bounded Spigot algorithm for transcendental constants

Rabinowitz-Wagon Spigot Algorithms for π

The spigot algorithm for calculating π was discovered by Rabinowitz-Wagon in 1990 See [12]. The formula is remarkable simple and does not required any fancy computing juts the basic operation like, add, subtract, multiply and divide and can be implemented using only integer arithmetic.
However, it still requires that in order to compute the n-digits of π, you still need to calculate all the receding n-1 digits.

The spigot algorithm is based on the expansion for π:

$$\pi=\sum_{n=0}^{\infty}\frac{(n!)^2 2^{n+1}}{(2n+1)!}$$

This series can be expanded into a Horner type schema.
To see that we can just run the first couple of expansion e.g. n=0,1,2:

$$\pi=\sum_{n=0}^{\infty}\frac{(n!)^2 2^{n+1}}{(2n+1)!}=\frac{1*2}{1!}+\frac{1*2^2}{3!}+\frac{(2!)^2*2^3}{5!}+\cdots=$$

$$2+2\frac{1*2}{2*3}+2\frac{2*2*2*2}{2*3*4*5}+\cdots=$$

$$2+2\frac{1}{3}+2\frac{1}{3}\frac{2}{5}+\cdots=2+\frac{1}{3}(2+\frac{2}{5}(2,\dots))$$

This series expands out using the Horner schema into:

$$\pi = 2 + \frac{1}{3}(2 + \frac{2}{5}\left(2 + \frac{3}{7}\left(\dots\left(2 + \frac{n}{2n+1}(\dots)\right)\right)\right))$$

This is known to be a mixed-radix base $c = \left(\frac{1}{3}, \frac{2}{5}, \frac{3}{7}, \frac{4}{9}, \dots\right)$ with respect to π=(2;2,2,2,..).

You can setup a simple excel sheet calculating the digits in π as the one below, se [12] for a detailed explanation of the formula in each cell. The π digit is showing up in the gray column below as 3.1415. Now the number of terms you would need to calculate n digits of the digits π is bound by $(\frac{10n}{3} + 1)$ see [13].

Spigot π																
	Terms	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
	A=	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
	B=		3	5	7	9	11	13	15	17	19	21	23	25	27	29
Initialize		2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
Scale		20	20	20	20	20	20	20	20	20	20	20	20	20	20	20
Carry	3	10	12	12	12	10	12	7	8	9	0	0	0	0	0	
Sum		30	32	32	32	30	32	27	28	29	20	20	20	20	20	20
remainders		0	2	2	4	3	10	1	13	12	1	20	20	20	20	20
Scale		0	20	20	40	30	100	10	130	120	10	200	200	200	200	200
Carry	1	13	20	33	40	65	48	98	88	81	170	165	156	130	84	
Sum		13	40	53	80	95	148	108	218	201	180	365	356	330	284	200
remainders		3	1	3	3	5	5	4	8	14	9	8	11	5	14	26
Scale		30	10	30	30	50	50	40	80	140	90	80	110	50	140	260
Carry	4	11	24	30	40	45	54	77	96	72	70	77	72	117	112	
Sum		41	34	60	70	95	104	117	176	212	160	157	182	167	252	260
remainders		1	1	0	0	5	5	0	11	8	8	10	21	17	9	28
Scale		10	10	0	0	50	50	0	110	80	80	100	210	170	90	280
Carry	1	5	6	15	36	35	36	84	80	90	120	154	120	104	126	
Sum		15	16	15	36	85	86	84	190	170	200	254	330	274	216	280
remainders		5	1	0	1	4	9	6	10	0	10	2	8	24	0	19
Scaler		50	10	0	10	40	90	60	100	0	100	20	80	240	0	190
Carry	5	6	8	21	44	60	48	56	24	63	50	99	132	39	84	
Sum		56	18	21	54	100	138	116	124	63	150	119	212	279	84	190

Thanks to Dik Winter and Achim Flammenkamp they publish a condense version in the C language version of the algorithm that produce four digits at a time using only integer arithmetic. That version was later on beautified by Gibbons and bought below. The algorithm is said to be bounded meaning that it requires the desired number of digits you want to calculate π to prior. Gibbons in [13] establish an equivalent unbounded algorithm that just procedure a steady stream of π digits. The algorithm below procedure 4 digits of π per iterations. The number 14 below is coming from the number of terms formula above: $(\frac{10n}{3}+1) = (\frac{10*4}{3}+1) = 14$

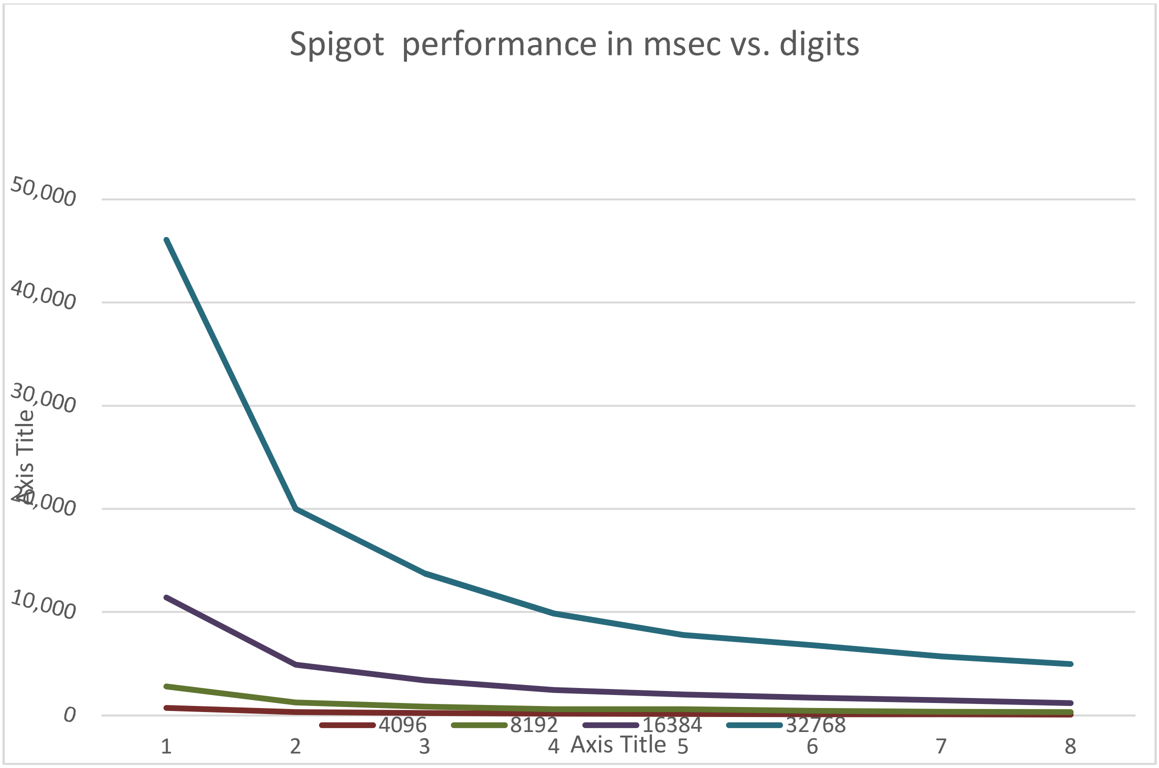
Algorithm 1.1 Spigot Gibbons

```
#define NDIGITS  15000              /*max.digits to compute*/
#define LEN      (NDIGITS/4+1)*14 /*nec.arraylength*/
Int a[LEN];                         /*arrayof4digit-decimals*/
Int b;                              /*nominatorprev.base*/
Int c=LEN;                          /*index*/
Int d=0;                            /*accumulatorandcarry*/
Int e=0;                            /*saveprev.4digits*/
Int f=10000;                        /*newbase,4dec.digits*/
Int g;                              /*denomprev.base*/
Int h=0;                            /*initswitch*/
//Spigotalgorithms 4
Int main(){
        for(;(b=c-=14)>0;)          /*outerloop:4digits/loop*/
        {
        for(;--b>0;)                /*innerloop:radixconv*/
        {
        d*=b;                       /*acc*=nom.prevbase*/
        if(h==0)
        d+=2000*f;              /*firstouterloop*/
        else
             d+=a[b]*f;             /*non-firstouterloop*/
        g=b+b-1;                    /*denomprev.base*/
        a[b]=d%g;
        d/=g;                       /*savecarry*/
        }
        h=printf("%04d",e+d/f);     /*printprev 4 digits*/
        d=e=d%f;                    /*savecurrent 4 digits*/
        }
        return0;
        }
```

The Algorithm above can deliver approx. 15,000 digits of π without going into overflow. Let us try to improve that and make it more useful to generate more number of π digits. The first improving we can do is to use 32bit unsigned integer arithmetic instead of signed that will take the above algorithm to handle 30,723 π digits before it goes into overflow. See below.

The biggest issue is overflow in the accumulator variable d. Since we for four digits at a time are initially multiplying the 2,000 with the constant F that is 10,000 and add it to the accumulator d, we add a digit that is of magnitude of $2*10^7$. The maximum digit that can be hold using 32bit unsigned integer arithmetic is ~ $4*10^9$ and that why the algorithm goes into overflow shortly after 30,000 digits of π has been calculated.

The next think we can do is to lower the number of digit to add for each iterations, instead of 4 digit at a time we can change to e.g. 3 which increase the number of digit for π to a maximum of 293,261 before we go into overflow. Continue down that road we can increase the digits for PI to ~2,8Million digit and further to more than 26 million digits if we only find one digit at a time. However, this is not without a time penalty. See picture below that show the speed in milliseconds as a function of how many digits of π you need to calculate. The test was performed on an I7 CPU with a quad processer and 2.6MHz clock frequency. If you follow the blue line (π with 32,767 digits), you can see that below four digits at a time you see and dramatic increase in time on the other hand if you increase the number of digits per iteration to eight you make the algorithm twice as fast. However, that is not possible with below algorithm that only use 32-bit integer arithmetic. The maximum number of digits it can handle is five digits at a time but that will limited the digits of π to 3,474 before it goes into overflow.



Gibbons version of the π has a flaw that is not exposed with four digits at a time with the limited number of digits it can generate but is visible with lowering the number of digits. That is an overflow in the printout of e+d/f in the statement

```
h=printf("%04d",e+d/f);
```

The issue is that it sometimes generate a carry that is not added to the π digit for the previous digit and therefore it failed to generate the correct result for π. Instead, we change it to accumulate the π digits into a std::string from the C++ standard template library. That way when a carry is detected we can propagate the carry back into the already calculated digit correctly.

The 32bit & 64bit version of the final algorithm is listed below.

32-bit version:

Algorithm 1.2 pi_spigot_32()

```
// 32bit version of the spigot algorithm.
//
std::string pi_spigot_32(const int digits, int no_dig = 4)
            {
            static unsigned long f_table[] = { 0, 10, 100, 1000, 10000, 100000 };
            static unsigned long f2_table[] = { 0,  2,  20,  200,  2000, 20000 };
            const int TERMS = (10 * no_dig / 3 + 1);
            bool first_time = true;                         // First time in loop flag
            bool overflow_flag = false;                     // Overflow flag
            char buffer[32];
            std::string ss;                                 // The String that hold the calculated PI

            long b, c;                                      // Loop counters
            int carry, no_carry = 0;                        // Outer loop carrier, plus no of carroer adjustment counts
            unsigned long f, f2;                            // New base 1 decimal digits at a time
            unsigned long dig_n = 0;                        // dig_n holds the next no_dig digit to add
            unsigned long e = 0;                            // Save previous 4 digits
            unsigned long acc = 0, g = 0, tmp32;
            ss.reserve(digits + 16);                        // Pre reserve the string size to be able to accumulate all digits plus 8
            if (no_dig > 5) no_dig = 5;                     // ensure no_dig<=5
            if (no_dig < 1) no_dig = 1;                     // Ensure no_dig>0
            c = (digits / no_dig + 1) * no_dig;             // Since we do collect PI in trunks of no_dig digit at a time we need to ensure
                                                            // digits is divisble by no_dig.
            if (no_dig == 1) c++;                           // Extra guard digit for 1 digit at a time.
            c = (c / no_dig + 1) * TERMS;                   // c ensure that the digits we seek is divisble by no_dig
            f = f_table[no_dig];                            // Load the initial f
            f2 = f2_table[no_dig];                          // Load the initial f2

            unsigned long *a = new unsigned long [c];       // Array of 4 digits decimals

            // b is the nominator previous base; c is the index
            for (; (b = c -= TERMS) > 0 && overflow_flag == false; first_time = false)
            {
                        for (; --b > 0 && overflow_flag == false;)
                        {           // Check for overflow
                                    if (acc > ULONG_MAX / b) overflow_flag = true;
                                    acc *= b;                           // Accumulator *= nom previous base
                                    tmp32 = f;
                                    if (first_time == true)             // Test for first run in the main loop
                                                tmp32 *= f2;            // First outer loop. a[b] is not yet initialized
                                    else
                                                tmp32 *= a[b];          // Non first outer loop. a[b] is initialized in the first loop
                                    if (acc > ULONG_MAX - tmp32) overflow_flag = true;          // Check for overflow
                                    acc += tmp32;                       // add it to accumulator
                                    g = b + b - 1;                      // denominated previous base
                                    a[b] = acc % g;                     // Update the accumulator
                                    acc /= g;                           // save carry
                        }
                        dig_n = (unsigned long)(e + acc / f);           // Get previous no_dig digits. Could occasinaly be no_dig+1 digits in
                                                                        // which case we have to propagate back the extra digit.
                        carry = (unsigned)(dig_n / f);                  // Check for extra carry that we need to propagate back into the
                                                                        // current sum of PI digits
                        dig_n %= f;                                     // Eliminate the extra carrier so now l contains no_dig digits to add
                                                                        // to the string

                        // Add the carrier to the existing number for PI calculated so far.
                        if (carry > 0)
                                    {
                                    ++no_carry;                         // Keep count of how many carrier detect
                                    // Loop and propagate back the extra carrier to the existing PI digits found so far
                                    for (int i = ss.length(); carry > 0 && i > 0; --i)
                                                {
                                                // Never seen more than one loop here but it can handle multiple carry back propagation
                                                int new_digit
                                                new_digit = (ss[i - 1] - '0') + carry;          // Calculate new digit
                                                carry = new_digit / 10;                         // Calculate new carry if any
                                                ss[i - 1] = new_digit % 10 + '0';   // Put the adjusted digit back in our PI digit list
                                                }
                        }

                        (void)sprintf(buffer, "%0*lu", no_dig, dig_n);  // Print previous no_dig digits to buffer
                        ss += std::string(buffer);                      // Add it to PI string
                        if (first_time == true)
                                    ss.insert(1, ".");                  // add the decimal pointafter the first digit to create 3.14...
                        acc = acc % f;                                  // save current no_dig digits and repeat loop
                        e = (unsigned long)acc;
                        }

            if(overflow_flag==false)
```

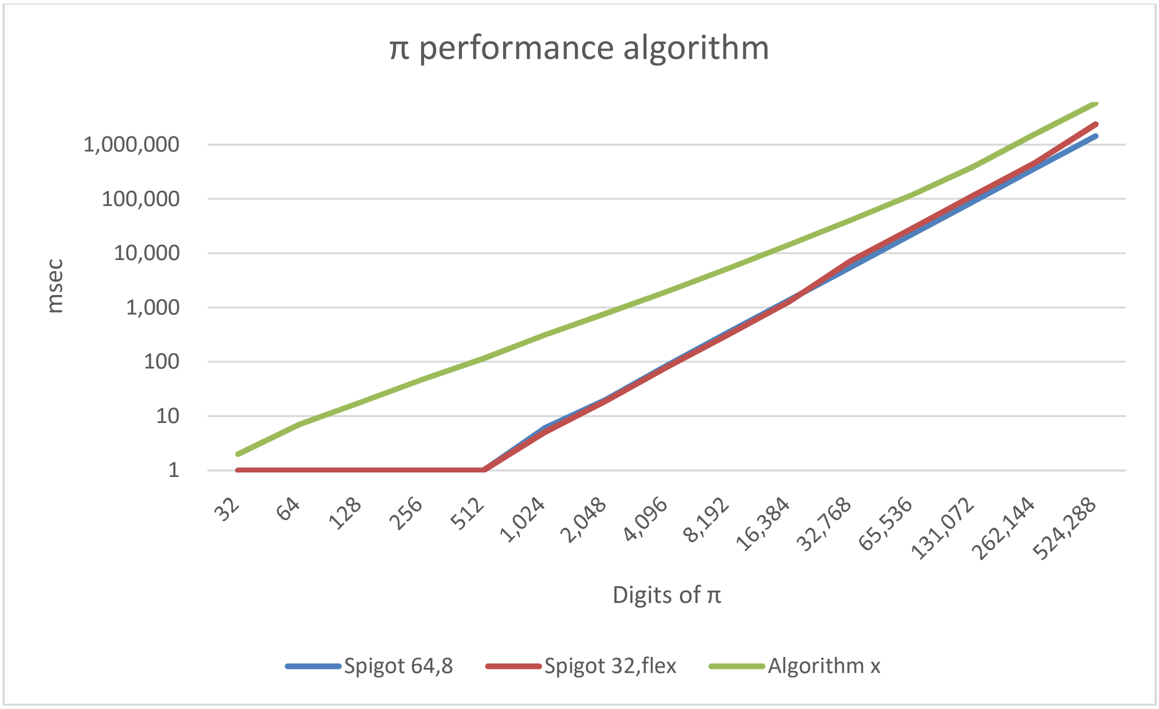
```
                        ss.erase(digits + 1);                             // Remove the extra digits that we didnt requested but used as guard
                                                                          // digits
            else
                        ss = std::string("Overflow:") + ss;               // Set overflow in the return string
            delete a;                                                     // Delete the a[];
            return ss;                                                    // Return Pi with the number of digits
            }
```

64-bit version

Algorithm 1.3 pi_spigot_64()

```
// 64bit version of the spigot algorithm.
// Notice acc, a, g needs to be unsigned 64bit.
// Emperisk for pi to 2^n digits, acc need to hold approx 2^(n+17) numbers. while a[] and g needs approx 2^(n+3)
// numbers
// a[] & g could potential be unsigned long (32bit) going to a max of 2^29 digit or 536millions digit of PI. but with
// unsigned 64bit you can "unlimited"
std::string pi_spigot_64( const int digits, int no_dig = 4 )
          {
          static unsigned long f_table[]= { 0, 10, 100, 1000, 10000, 100000, 1000000, 10000000, 100000000 };
          static unsigned long f2_table[] = { 0,  2,  20,  200,  2000,  20000,  200000,  2000000,  20000000 };
          const int TERMS = (10 * no_dig / 3 + 1);
          bool first_time = true;                         // First time in loop flag
          bool overflow_flag = false;                     // Overflow flag
          char buffer[32];
          std::string ss;                                 // The String that hold the calculated PI
          long b, c;                                      // Loop counters
          int carry, no_carry = 0;                        // Outer loop carrier, plus no of carroer adjustment counts
          unsigned long f, f2;                            // New base 1 decimal digits at a time
          unsigned long dig_n = 0;                        // dig_n holds the next no_dig digit to add
          unsigned long e = 0;                            // Save previous no_dig digits
          unsigned _int64 acc = 0, g = 0, tmp64;
          ss.reserve(digits + 16);      // Pre reserve the string size to be able to accumulate all digits plus 8
          if (no_dig > 8) no_dig = 8;                     // ensure no_dig<=8
          if (no_dig < 1) no_dig = 1;                     // Ensure no_dig>0
          c = (digits / no_dig + 1) * no_dig;             // Since we do collect PI in trunks of no_dig digit at a
                                                          // time we need to ensure digits is divisble by no_dig.
          if (no_dig == 1) c++;                           // Extra guard digit for 1 digit at a time.
          c = (c / no_dig + 1) * TERMS;                   // c ensure that the digits we seek is divisble by no_dig
          f = f_table[no_dig];                            // Load the initial f
          f2 = f2_table[no_dig];                          // Load the initial f2

          unsigned _int64 *a = new unsigned _int64[c];    // Array of 4 digits decimals
          // b is the nominator previous base; c is the index
          for (; (b = c -= TERMS) > 0 && overflow_flag==false; first_time=false)
                    {
                    for (; --b > 0 && overflow_flag==false;)
                              {
                              if (acc > ULLONG_MAX / b) overflow_flag = true; // Check for overflow
                              acc *= b;                   // Accumulator *= nom previous base
                              tmp64 = f;
                              if (first_time==true)       // Test for first run in the main loop
                                        tmp64 *= f2;      // First outer loop. a[b] is not yet initialized
                              else
                                        tmp64 *= a[b];    // Non first outer loop. a[b] is initialized in the first
                                                          // loop
                              if (acc > ULLONG_MAX - tmp64) overflow_flag = true; // Check for overflow
                              acc += tmp64;               // add it to accumulator
                              g = b + b - 1;              // denominated previous base
                              a[b] = acc % g;             // Update the accumulator
                              acc /= g;                   // save carry
                              }
                    dig_n =(unsigned long)( e + acc / f );  // Get previous no_dig digits. Could occasinaly be no_dig+1
                                                          // digits in which case we have to propagate back the extra
                                                          // digit.
                    // Check for extra carry that we need to propagate back into the current sum of PI digits
                    carry = (unsigned)( dig_n / f);
                    dig_n %= f;                           // Eliminate the extra carrier so now l contains no_dig
                                                          // digits to add to the string
                    // Add the carrier to the existing number for PI calculated so far.
                    if (carry > 0)
                              {
                              ++no_carry;                 // Keep count of how many carrier detect
                              // Loop and propagate back the extra carrier to the existing PI digits found so far
                              for (int i = ss.length(); carry > 0 && i > 0; --i)
                                        {
                                        // Never seen more than one loop here but it can handle multiple carry back
                                        // propagation
                                        int new_digit;
                                        new_digit = (ss[i - 1] - '0') + carry;  // Calculate new digit
                                        carry = new_digit / 10;                 // Calculate new carry if any
                                        ss[i - 1] = new_digit % 10 + '0';       // Put the adjusted digit back in our
                                                                                // PI digit list
                                        }
                              }

                    (void)sprintf(buffer, "%0*lu", no_dig, dig_n);    // Print previous no_dig digits to buffer
                    ss += std::string(buffer);                        // Add it to PI string
                    if(first_time==true)
                              ss.insert(1, ".");  // add the decimal pointafter the first digit to create 3.14...
                    acc = acc % f;                // save current no_dig digits and repeat loop
                    e = (unsigned long)acc;
                    }

          ss.erase(digits+1);           // Remove the extra digits that we didnt requested but used as guard digits
          if (overflow_flag == true)
                    ss = std::string("Overflow:") + ss;      // Set overflow in the return string
          delete a;                                          // Delete the a[];

          return ss;                                         // Return Pi with the number of digits
          }
```

Time comparison is that the 32bit is faster in the range of π digits that both algorithm can handle. This is not a surprise since 64-bit integer arithmetic is more time consuming that the equivalent 32bit integer arithmetic.



Note: The algorithm x is referring to calculating π using arbitrary precision arithmetic and the Borwein algorithm, see my paper on that subject.

Gosper Algorithms for π

As it has been mention in the previous section, Rabinowitz-Wagon Spigot Algorithms for π requires approximately 3.3 terms per digits of π. However, Gosper page formula for π can also be used and it is more efficient since it requires less terms to be evaluated per digits. The number of digits you get is approx. 1.1 digit or if you evaluate 10 terms you get 11 valid digits of π. This is approx. 3 times less work to perform per digits, however as always you do not get things for free. Each terms is a little bit more complicated to handle and you quicker reach the limit of the integer representation so for all practical purpose you need to implement this algorithm using 64-bit integer arithmetic only.

The Gosper formula is:

$$\pi = 3 + 2\sum_{n=1}^{\infty} \frac{n(5n+3)(2n-1)!\,(n!)}{2^{n-1}(3n+2)!}$$

Which expand into this series:

$$\pi = 3 + \frac{1}{60}(8 + \frac{2\times 3}{7\times 8\times 3}\left(13 + \frac{3\times 5}{10\times 11\times 3}\left(18 + \frac{4*7}{13*14*3}(\dots)\right)\right))$$

And:

$$\pi = 3 + \frac{1}{60}(8 + \frac{6}{168}\left(13 + \frac{15}{330}\left(18 + \frac{28}{816}\left(5n-2 + \frac{n(2n-1)}{3(9(n^2+n)+2)}(\dots)\right)\right)\right))$$

This is the way we want to have the series expanded so we can quickly identifies the different Spigot elements. This is well-known mixed-radix base $c = \left(\frac{1}{60}, \frac{6}{168}, \frac{15}{330}, \frac{28}{816}, \frac{n(2n-1)}{3(9(n^2+n)+2)}, \dots\right)$ with respect to π=(3;8,13,18,5n-2,..).

As the fraction or two terms is always smaller than $\frac{1}{13}$ you would get d precision with d= $\frac{\log(10^n)}{\log(13)} => d \approx \frac{n}{0.9}$ terms.

The new simple excel sheet calculating the digits in π as the one below, se [14] for a detailed explanation of the formula in each cell. The π digit is showing up in the gray column below as 3.1415. Now the number of terms you would need to calculate n digits of the digits π is bound by digits/0.9 see [16]. In the below table we see that we only need 6 terms to get approximately seven correct digits which is a lot less that Rabinowitz-Wagon algorithm. However you also notice that the mixed radix based $\frac{A}{B}$ quickly get into some high numbers that can cause overflow if not carefully managed.

Spigot π - Gosper								
	Terms		0	1	2	3	4	5
	A			1	6	15	28	45
	B			60	168	330	546	816
Initialize			3	8	13	18	23	28
Scale			30	80	130	180	230	280
Carry		**3**	1	0	0	0	0	
Sum			31	80	130	180	230	280

remainders		1	20	130	180	230	280
Scale		10	200	1300	1800	2300	2800
Carry	**1**	4	48	75	112	135	
Sum		14	248	1375	1912	2435	2800
remainders		4	8	31	262	251	352
Scale		40	80	310	2620	2510	3520
Carry	**4**	1	12	120	112	180	
Sum		41	92	430	2732	2690	3520
remainders		1	32	94	92	506	256
Scale		10	320	940	920	5060	2560
Carry	**1**	5	30	45	252	135	
Sum		15	350	985	1172	5195	2560
remainders		5	50	145	182	281	112
Scaler		50	500	1450	1820	2810	1120
Carry	**5**	9	54	75	140	45	
Sum		59	554	1525	1960	2855	1120
remainders		9	14	13	310	125	304
Scaler		90	140	130	3100	1250	3040
Carry	**9**	2	6	135	56	135	
Sum		92	146	265	3156	1385	3040
remainders		2	26	97	186	293	592
Scaler		20	260	970	1860	2930	5920
Carry	**2**	4	36	90	140	315	
Sum		24	296	1060	2000	3245	5920

With only six terms, we get seven correct digits of π (3.141592). Only drawbacks with Gosper algorithm over the Rabinowitz-Wagon Spigot Algorithms for π is that the mixed radix based $\left(\frac{1}{60},\frac{6}{168},\frac{15}{330},\frac{28}{816},\dots,\frac{n(2n-1)}{3(9(n^2+n)+2)},\dots\right)$ yield higher that the Rabinowitz-Wagon algorithm that used $\left(\frac{1}{3},\frac{2}{5},\frac{3}{7},\frac{4}{9},\dots,\frac{n}{2n+1}\right)$. This leads to faster overflow even when using 64-bit integer arithmetic. On the other hand, we do not needs as many terms as the Rabinowitz-Wagon Algorithm. For a wanted precision of *d* digits, we need for the Rabinowitz-Wagon algorithm n=($\frac{10d}{3}+1$). For the Gosper algorithm, we need $n\sim\frac{d}{0.9}$. Dividing the two formula you get ratio $\sim 3+\frac{0.9}{d}$=> or 3 for larger number of d. e.g lets assume you need to find 1,000,000 digits precision of π. The largest term need for Gosper even with the reduced number of terms is $\frac{\sim 6.67^{11}}{\sim 9^{12}}$ while for Rabinowitz-wagon it is $\frac{10^6}{\sim 2x10^6}$.

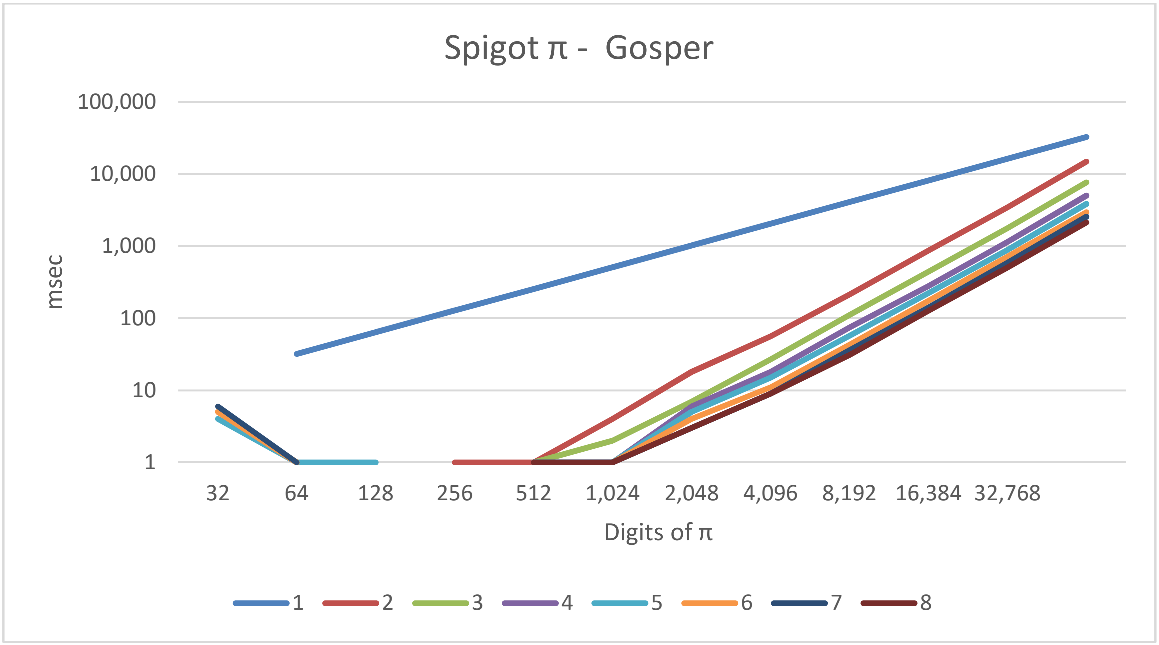
Clearly, we need to expect the Gosper algorithm to overflow faster for large *d* than the Rabinowitz-Wagon algorithm.

64-bit version:

Algorithm 2.2 pi_spigot_gosper_64()

```
// Gosper algorithm
// A Column: 1,6,15,28,45,...  2n(n-1)-n
// B Column: 60, 168, 330,  546,  816,... 3(9(n+1)n+2)
// Initialization values:3, 8, 13, 18, 23 28,... 5n-2
std::string pi_spigot_gosper_64(int digits, int no_dig = 1)
        {
        static unsigned long f_table[] = { 1, 10, 100, 1000, 10000, 100000, 1000000, 10000000,
100000000 };
        bool first_time = true;
        bool overflow_flag = false;
        char buffer[32];
        std::string ss;
        int dig;
        unsigned int car, no_carry = 0;
        unsigned int no_terms;          // No of terms to complete as a function of digits
        unsigned long f, f2;            // New base 1 decimal digits at a time
        unsigned long dig_n;            // dig_n holds the next no_dig digit to add
        unsigned _int64 carry, a, b, tmp64;
        ss.reserve(digits + 16);

        if (no_dig > 8) no_dig = 8;     // ensure no_dig<=5
        if (no_dig < 1) no_dig = 1;     // Ensure no_dig>0
        // Since we do it in trunks of no_dig digits at a time we need to ensure digits is
divisble with no_dig.
        dig = (digits / no_dig + (digits%no_dig>0 ? 1 : 0)) * no_dig;
        dig += no_dig;                  // Extra guard digits
        no_terms = (unsigned int)(dig * 0.9) + 1; // Calculate the number of terms needed
        unsigned _int64 *acc = new unsigned _int64[no_terms + 1];  // Allocate the needed
accumulator
        f = f_table[no_dig];            // Load the initial f
        f2 = f_table[no_dig - 1];       // Load the initial f2

        for (int i = dig; i >= 0 && overflow_flag == false; i -= no_dig, first_time = false)
                {
                carry = 0;
                no_terms = (unsigned int)(i * 0.9) + 1; // Calculate the number of terms needed
                for (int j = no_terms; j>0 && overflow_flag == false; --j)
                        {
                        a = 2 * (j + 1) - 1;     // Create Column A terms
                        a *= (j + 1);   // Take the previous column A and multiply it with carry
                        if (carry > ULLONG_MAX / a)
                                overflow_flag = true;    // Check for overflow
                        carry *= a;
                        tmp64 = f;
                        if (first_time == true)
                                {
                                tmp64 *= f2;
                                tmp64 *= (5 * (j + 1) - 2);      // Create the initialized value
                                }
                        else
                                tmp64 *= acc[j];
                        if (carry > ULLONG_MAX - tmp64)
                                overflow_flag = true;
                        carry += tmp64;
                        b = j;          //Assign it to 64bit variable b to avoid 32bit overflow.
                        b = 3 * (9 * (b + 1)*b + 2);             // Create Column B terms
                        acc[j] = carry % b;
                        carry /= b;
                        }

                if (first_time == true)
                        {
                        tmp64 = f; tmp64 *= 3 * f2;
                        acc[0] = (tmp64 + carry);
                        }
                else
                        acc[0] = acc[0] * f + carry;
                dig_n = (unsigned)(acc[0] / f);
                car = (unsigned)(dig_n / f);
                dig_n %= f;
                // Add the carry to the existing number for PI calculated so far.
                if (car > 0)
                        {
                        ++no_carry;
                        for (int j = ss.length(); car > 0 && j > 0; --j)
                                {
                                int dd;
                                dd = (ss[j - 1] - '0') + car;
                                car = dd / 10;
                                ss[j - 1] = dd % 10 + '0';
                                }
                        }
                (void)sprintf(buffer, "%0*lu", no_dig, dig_n);
                ss += std::string(buffer);
                acc[0] %= f;
                }

        ss.insert(1, ".");// add a come after the first digit to create 3.14...
        if (overflow_flag == false)
                ss.erase(digits + 1); // Remove the extra digits that we didnt requested.
        else
                ss = std::string("Overflow:") + ss;
        delete acc;
        return ss;
        }
```

The above mention algorithm can find π and for each loop, we can find between one and eight digits. Not surprisingly, the more digits we find per loop the faster the overall algorithm is as showed in below diagram. The number 1 to 8 refer to how many digits we find per loop and in calculation, π with digits from 32 to 32,768 digits and the timing on the Y-axis is milliseconds.



Notice the scale is logarithm, to find π eight digits at a time is approximately 10 times faster that applying the algorithm one digit at a time

Spigot Algorithm for e

An Algorithm for calculation of *e* to an arbitrary precision limit was publish back in 1967 by Sale [2]. It devised a spigot algorithm for the calculation of the transcendental number *e*. The original article listed an Algol60 source program for the calculation and I took the liberty to convert it to C++ and added a few improvements. The *e* can be evaluated by the infinite series

$$e = \frac{1}{0!} + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \cdots$$

Alternatively, in another way as:

$$e = 1 + 1(1 + \frac{1}{2}\left(1 + \frac{1}{3}\left(1 + \cdots\left(1 + \frac{1}{n}\right) \ldots\right)\right))$$

Except for the two first term all other are less than one and we can further rewrite is as:

$$e = 2 + \frac{1}{2}(1 + \frac{1}{3}\left(1 + \frac{1}{4}(1 + \cdots)\right))$$

We can now create our usual Spigot table for *e* as shown below: e is showing up in the grey column as 2.71828182845

Spigot e

	Terms	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
	A		1	1	1	1	1	1	1	1	1	1	1	1	1	1
	B	10	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Initialize		1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
Scale		10	10	10	10	10	10	10	10	10	10	10	10	10	10	10
Carry	**2**	7	4	3	2	1	1	1	1	1	0	0	0	0	0	
Sum		17	14	13	12	11	11	11	11	11	10	10	10	10	10	10
remainders		7	0	1	0	1	5	4	3	2	0	10	10	10	10	10
Scale		70	0	10	0	10	50	40	30	20	0	100	100	100	100	100
Carry	**7**	1	3	0	3	9	6	4	2	0	9	9	8	7	6	
Sum		71	3	10	3	19	56	44	32	20	9	109	108	107	106	100
remainders		1	1	1	3	4	2	2	0	2	9	10	0	3	8	10
Scale		10	10	10	30	40	20	20	0	20	90	100	0	30	80	100
Carry	**1**	8	6	9	8	3	2	0	3	9	9	0	2	6	6	
Sum		18	16	19	38	43	22	20	3	29	99	100	2	36	86	100
remainders		8	0	1	2	3	4	6	3	2	9	1	2	10	2	10
Scale		80	0	10	20	30	40	60	30	20	90	10	20	100	20	100
Carry	**8**	2	5	6	7	8	9	4	3	9	1	2	7	1	6	
Sum		82	5	16	27	38	49	64	33	29	91	12	27	101	26	100
remainders		2	1	1	3	3	1	1	1	2	1	1	3	10	12	10
Scaler		20	10	10	30	30	10	10	10	20	10	10	30	100	120	100
Carry	**2**	8	6	9	6	1	1	1	2	1	1	3	8	9	6	
Sum		28	16	19	36	31	11	11	12	21	11	13	38	109	126	100
remainders		8	0	1	0	1	5	4	4	3	1	2	2	5	0	10
Scaler		80	0	10	0	10	50	40	40	30	10	20	20	50	0	100
Carry	**8**	1	3	0	3	9	6	5	3	1	1	1	3	0	6	
Sum		81	3	10	3	19	56	45	43	31	11	21	23	50	6	100
remainders		1	1	1	3	4	2	3	3	4	1	10	11	11	6	10
Scaler		10	10	10	30	40	20	30	30	40	10	100	110	110	60	100
Carry	**1**	8	6	9	8	4	4	4	4	1	9	9	8	4	6	
Sum		18	16	19	38	44	24	34	34	41	19	109	118	114	66	100
remainders		8	0	1	2	4	0	6	2	5	9	10	10	10	10	10
Scaler		80	0	10	20	40	0	60	20	50	90	100	100	100	100	100
Carry	**8**	2	5	7	8	1	9	3	6	9	9	9	8	7	6	
Sum		82	5	17	28	41	9	63	26	59	99	109	108	107	106	100

remainders		2	1	2	0	1	3	0	2	5	9	10	0	3	8	10
Scaler		20	10	20	0	10	30	0	20	50	90	100	0	30	80	100
Carry	**2**	8	6	0	3	5	0	3	6	9	9	0	2	6	6	
Sum		28	16	20	3	15	30	3	26	59	99	100	2	36	86	100

remainders		8	0	2	3	0	0	3	2	5	9	1	2	10	2	10
Scaler		80	0	20	30	0	0	30	20	50	90	10	20	100	20	100
Carry	**8**	4	9	7	0	0	4	3	6	9	1	2	7	1	6	
Sum		84	9	27	30	0	4	33	26	59	91	12	27	101	26	100

remainders		4	1	0	2	0	4	5	2	5	1	1	3	10	12	10
Scaler		40	10	0	20	0	40	50	20	50	10	10	30	100	120	100
Carry	**4**	5	1	5	1	7	7	3	5	1	1	3	8	9	6	
Sum		45	11	5	21	7	47	53	25	51	11	13	38	109	126	100

remainders		5	1	2	1	2	5	4	1	6	1	2	2	5	0	10
Scaler		50	10	20	10	20	50	40	10	60	10	20	20	50	0	100
Carry	**5**	8	7	3	5	9	6	2	6	1	1	1	3	0	6	
Sum		58	17	23	15	29	56	42	16	61	11	21	23	50	6	100

As can be seen you get 12 correct digit using the first 15 terms of the series expansion listed in vertical in bold in column two. Now we need to figure out how many terms we would need for a giving number of wanted digits, *d*. If the last term of $\frac{1}{n!}$ is less than $10^{-(d+1)}$ where *d* is the number of digits wanted then we can stop. To avoid overflow we use Stirling approximation formula for $n! \sim \sqrt{2\pi d}(\frac{d}{e})^d$ and take the log() on both side to get:

$$\frac{1}{n!} < \frac{1}{10^{d+1}} =>$$

$$n! < 10^{d+1} =>$$

Now take log() on both side and you get:

$$n(\log(n) - 1) + \frac{1}{2}\log(2\pi d) < (d+1)\log(10)$$

In order to solve n for a giving number of digits, d we use the Newton iteration that quickly finds n in typical 4-5 iterations.

Newton formula:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} =>$$

And substituting f(x) and f'(x) in you get:

$$x_{n+1} = x_n - \frac{x_n((\log(x_n) - 1) + \frac{1}{2}\log(2\pi d) - (d+1)log(10)}{\frac{1}{2x_n} + \log(x_n)}$$

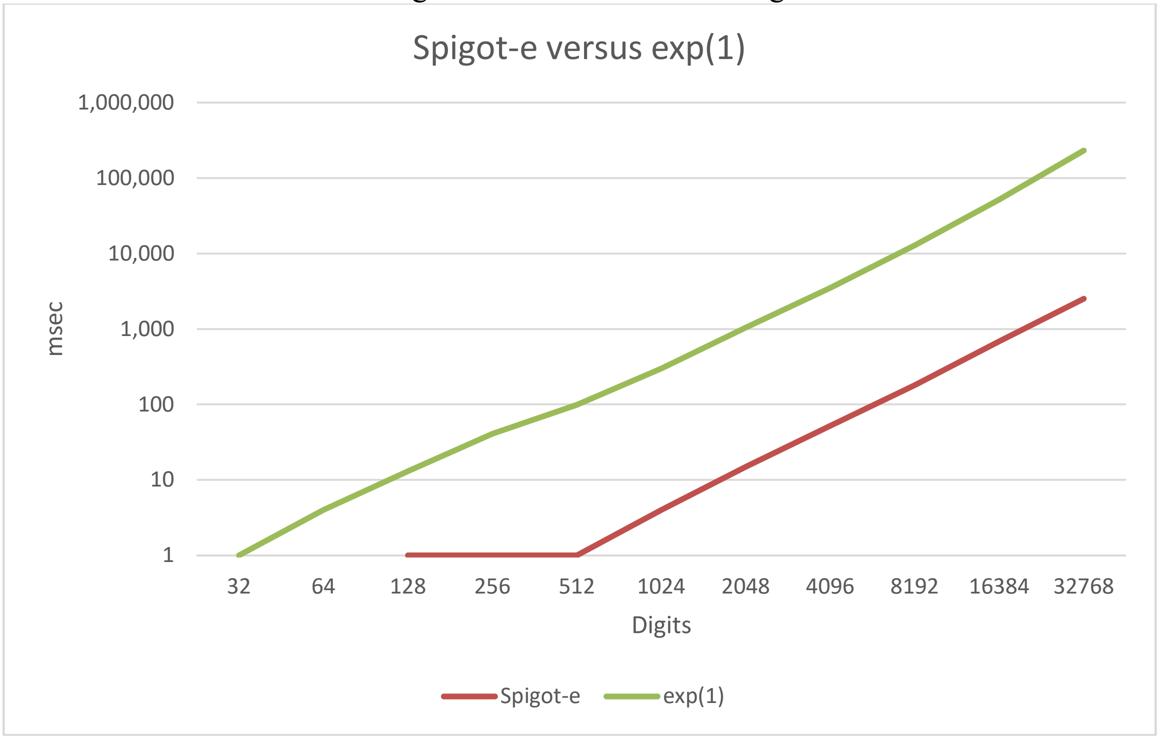
The number of terms for 10 digits precisions are 15 terms; for 100 digits precision 71 terms and for 1000 digits precision is 451 terms just to give you an idea of what we are expecting as we scale the number of digits for *e*.

Algorithm 3.1 spigot_e()

```
// Spigot algorithm for e
// From The computer Journal 1968 (A H J Sale) written in Algo 60 and ported to
// c++ with some modifications
std::string spigot_e(int digits)
       {
       unsigned int m;
       unsigned int tmp,  carry;
       double test = (digits + 1) * log(10);
       bool first_time = true;
       unsigned int *coef;
       std::string ss("2.");
       ss.reserve(digits + 16);
       double xnew, xold;

       // Stirling approximation of m!~Sqrt(2*pi*digits)(digits/e)^digits.
       // Taken log on both side you get:
       // m*(Math.log((m)-1)+0.5*Math.log(2*Math.pi*m);
       // Use Newton method to find x in less that 4-5 iterations
       for (xold = 5, xnew = 0; ; xold=xnew )
              {
              Double f = xold*(log(xold)-1)+0.5*log(2*3.141592653589793*xold);
              double f1 = 0.5 / xold + log(xold);
              xnew = xold - (f - test) / f1;
              if ((int)ceil(xnew) == (int)ceil(xold))
                     break;
              }
       m = (unsigned int)ceil(xnew);
       if (m < 5)
              m = 5;
       coef = new unsigned int[m+1];

       // Loop for each digit
       for ( int i = 1; i <= digits; ++i, first_time = false)
              {
              carry = 0;
              for ( int j = m; j >= 2; j--)
                     {
                     if (first_time == true)
                            tmp = 10;
                     else
                            tmp = coef[j] * 10;
                     tmp += carry;
                     carry = tmp / (j);
```

```
                        coef[j] = tmp % (j);
                        }
                ss.append( 1, (char)(carry+'0') );
                }
        delete coef;
        return ss;
        }
```

Performance is outstanding compare to regular calculation using exp(1) as a Taylor series.
See chart below. Spigot-e outperformed the traditional algorithm with a factor of 70-90 that factor increases the more digits we need above 32767digits



Spigot Algorithm for ln(2)

Let us turn our attention to the ln(2) another transcendental constant. The series expansion for ln(2) is $\sum_{n=1}^{\infty}\frac{1}{2^n n}$. As usually, we need to rewrite the series expansion into a Horner type representation and you get:

$$\ln(2) = \sum_{n=1}^{\infty}\frac{1}{2^n n} =>$$

$$\ln(2) = \frac{1}{2} + \frac{1}{2^2 2} + \frac{1}{2^3 3} + \frac{1}{2^4 4} + \frac{1}{2^5 5} + \cdots =>$$

$$\ln(2) = \left(\frac{1}{2} + \frac{1}{2}\left(\frac{1}{2^2} + \frac{1}{2^2 3} + \frac{1}{2^3 4} + \frac{1}{2^4 5} + \cdots\right)\right) =>$$

$$\ln(2) = \left(\frac{1}{2} + \frac{1}{2}\left(\frac{1}{4} + \frac{1}{2}\left(\frac{1}{2^1 3} + \frac{1}{2^2 4} + \frac{1}{2^3 5} + \cdots\right)\right)\right) \Rightarrow$$

$$\ln(2) = \left(\frac{1}{2} + \frac{1}{2}\left(\frac{1}{4} + \frac{1}{2}\left(\frac{1}{6} + \frac{1}{2}\left(\frac{1}{2^1 4} + \frac{1}{2^2 5} + \cdots\right)\right)\right)\right) \Rightarrow$$

$$\ln(2) = \left(\frac{1}{2} + \frac{1}{2}\left(\frac{1}{4} + \frac{1}{2}\left(\frac{1}{6} + \frac{1}{2}\left(\frac{1}{8} + \frac{1}{2}\left(\frac{1}{2^1 5} + \cdots\right)\right)\right)\right)\right) \Rightarrow$$

$$\ln(2)=\left(\frac{1}{2} + \frac{1}{2}\left(\frac{1}{4} + \frac{1}{2}\left(\frac{1}{6} + \frac{1}{2}\left(\frac{1}{8} + \frac{1}{2}\left(\frac{1}{10} + \cdots\right)\right)\right)\right)\right)$$

This is the way we want to have the series expanded so we can quickly identifies the different Spigot elements. This is a mixed-radix base $c = \left(\frac{1}{2}, \frac{1}{4}, \frac{1}{6}, \frac{1}{8}, \frac{1}{2n}, \ldots\right)$ with respect to ln(2)=($\frac{1}{2}$; $\frac{1}{2}$, $\frac{1}{2}$, $\frac{1}{2}$, ….). Only change is that we see we have a fraction $\frac{1}{2}$ as the initialized value and not a whole number, compare to the other algorithm presented here. The table formula will still work in principle but we ran into a problem using floating point arithmetic since we introduce rounding errors in our calculation and sure enough the table formula will only work correctly for up to 18 digits of ln(2) where after we get incorrect digits for ln(2). To fix this issue we need to alter the table formula to accommodate working with the correct fraction and carry it through our calculations.

The spigot table will now look like the above for finding ln(2) digits. And the second column is the result with 15 terms giving ln(2)=0.69314

Spigot LN2																	
	Terms	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	A		1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
	B	10	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
	2(n+1)	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30	32
Init N		1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
Init DN		2	4	6	8	10	12	14	16	18	20	22	24	26	28	30	32
Scale N		10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10
Carry	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Sum N		12	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10
Sum DN		2	4	6	8	10	12	14	16	18	20	22	24	26	28	30	32
Remainder N		6	1	5	5	1	5	5	5	5	1	5	5	5	5	1	5
Remainder DN		1	2	3	4	1	6	7	8	9	2	11	12	13	14	3	16
Scale N		60	10	50	50	10	50	50	50	50	10	50	50	50	50	10	50
Carry	6	9	13	10	8	7	6	5	4	4	3	3	2	2	2	1	0
Sum N		69	36	80	82	17	86	85	82	86	16	83	74	76	78	13	50

Sum DN		1	2	3	4	1	6	7	8	9	2	11	12	13	14	3	16
Remainder N		9	0	2	1	1	1	1	1	14	0	17	1	24	11	1	9
Remainder DN		1	1	3	2	1	3	7	4	9	1	11	6	13	7	3	8
Scale N		90	0	20	10	10	10	10	10	140	0	170	10	240	110	10	90
Carry	9	2	5	5	6	3	3	6	10	5	11	7	13	9	4	5	0
Sum N		92	5	35	22	13	19	52	50	185	11	247	88	357	138	25	90
Sum DN		1	1	3	2	1	3	7	4	9	1	11	6	13	7	3	8
Remainder N		2	1	5	1	1	1	10	1	5	1	5	2	19	12	1	5
Remainder DN		1	1	3	1	1	3	7	2	9	1	11	3	13	7	3	4
Scale N		20	10	50	10	10	10	100	10	50	10	50	20	190	120	10	50
Carry	3	11	12	9	8	6	9	5	6	8	6	9	12	10	4	6	0
Sum N		31	22	77	18	16	37	135	22	122	16	149	56	320	148	28	50
Sum DN		1	1	3	1	1	3	7	2	9	1	11	3	13	7	3	4
Remainder N		1	0	5	0	0	1	9	1	14	0	17	2	8	8	4	1
Remainder DN		1	1	3	1	1	3	7	1	9	1	11	3	13	7	3	2
Scale N		10	0	50	0	0	10	90	10	140	0	170	20	80	80	40	10
Carry	1	4	8	1	3	7	11	10	10	5	10	6	7	9	7	2	0
Sum N		14	8	53	3	7	43	160	20	185	10	236	41	197	129	46	10
Sum DN		1	1	3	1	1	3	7	1	9	1	11	3	13	7	3	2
Remainder N		4	0	5	1	1	1	6	0	5	0	16	5	15	3	4	1
Remainder DN		1	1	3	1	1	3	7	1	9	1	11	3	13	7	3	1
Scale N		40	0	50	10	10	10	60	0	50	0	160	50	150	30	40	10
Carry	4	6	12	8	7	4	5	2	5	6	13	12	8	6	9	5	0
Sum N		46	12	74	17	14	25	74	5	104	13	292	74	228	93	55	10
Sum DN		1	1	3	1	1	3	7	1	9	1	11	3	13	7	3	1

Now we need to figure out how many terms we would need for a giving number of wanted digits, *d*. if we see the fraction between two terms in the series expansion we get

$$\frac{\frac{1}{(n+1)2^{n+1}}}{\frac{1}{n2^{n}}} = \frac{n}{2(n+1)} = \frac{1}{2+\frac{1}{n}}$$

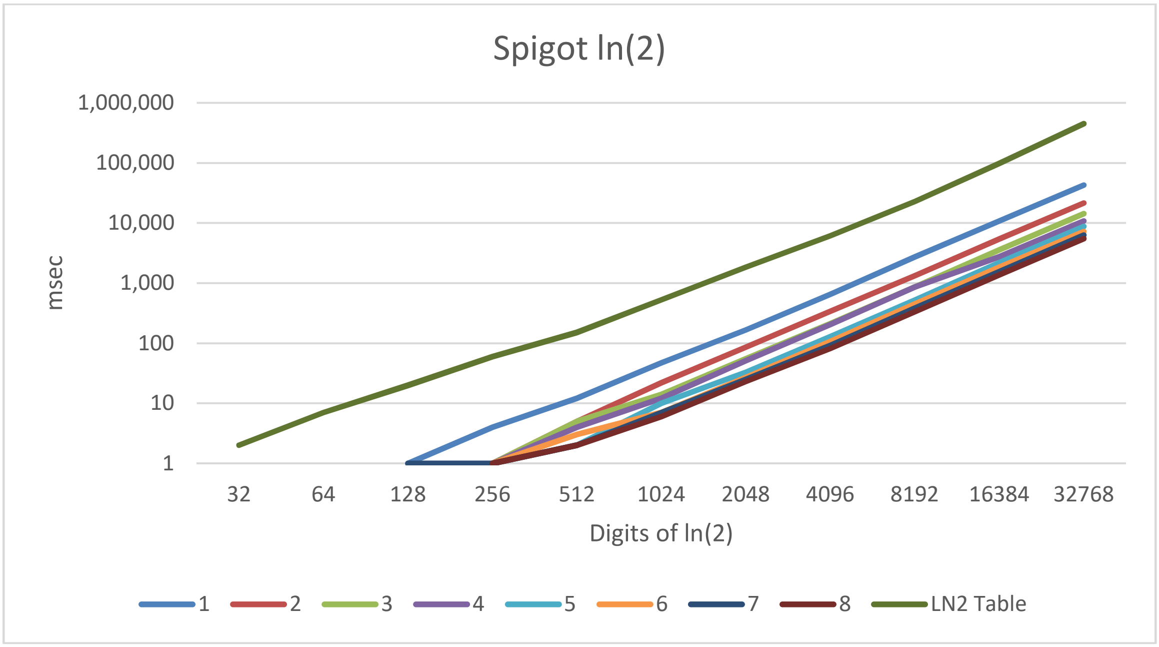
Or a factor slightly less than half the previous term. The is interesting enough the same reduction in terms as for the Rabinowitz-Wagon algorithm that yield the number of terms you would need to calculate for n digits of the ln(2) is bound by $(\frac{10n}{3}+1)$.
We are now ready to present the algorithm 4.1 for the calculation of ln(2)

Algorithm 4.1 spigot_ln2_64()

```
// 64 bit version of spigot algorithm for LN2
// It has automatic 64bit integer overflow detection in which case the result
// start with the string "Overflow...."
// A Column: 1,1,1,...,1
// B Column: 2,2,2,2,2,...,2
// Initialization values:0.5,0.25,...,1/(2*2^n)
std::string spigot_ln2_64(int digits, int no_dig = 1)
       {
       static unsigned long f_table[] =
{1,10,100,1000,10000,100000,1000000,10000000,100000000};
       bool first_time = true;     // First iteration of the algorithm
       bool overflow_flag = false; // 64bit integer overflow flag
       char buffer[32];
       std::string ss;             // The std::string that holds the ln2
       int dig;
       unsigned int car, no_carry = 0;
       unsigned int no_terms;      // No of terms to complete as a function of
digits
       unsigned long f;            // New base 1 decimal digits at a time
       unsigned long dig_n;        // dig_n holds the next no_dig digit to add
       unsigned long carry;
       unsigned _int64 tmp_n, tmp_dn;
       ss.reserve(digits + 16);

       if (no_dig > 8) no_dig = 8; // ensure no_dig<=8
       if (no_dig < 1) no_dig = 1; // Ensure no_dig>0
       // Since we do it in trunks of no_dig digits at a time we need to ensure
digits is divisble with no_dig.
       dig = (digits / no_dig + (digits%no_dig>0 ? 1 : 0)) * no_dig;
       dig += no_dig;              // Extra guard digits
       no_terms = (unsigned int)(10 * dig / 3 + 3);    // Calculate the number of
terms needed
       // Allocate the needed accumulators
       unsigned _int64 *acc_n = new unsigned _int64[no_terms + 1];
       unsigned _int64 *acc_dn = new unsigned _int64[no_terms + 1];
       f = f_table[no_dig];               // Load the initial f
       carry = 0;                         // Set carry to 0
       //Loop for each no_dig
       for (int i = dig; i >= 0 && overflow_flag == false; i -= first_time == true
? 1 : no_dig, first_time = false)
              {
              // Calculate new number of terms needed
              no_terms = (unsigned int)(10 * i / 3 + 3);
              // Loop for each no_terms
              for (int j = no_terms; j>0 && overflow_flag == false; --j)
                     {
                     if (first_time == true)
                            {// Calculate the initialize value
                            tmp_dn = (j + 1) * 2;
                            tmp_n = 1;
                            }
                     else
                            {
                            tmp_n = acc_n[j];
                            tmp_dn = acc_dn[j];
                            }
                     tmp_n *= f;           // Scale it
                     // Check for 64bit overflow. Not very likely
                     if (carry > 0 && tmp_dn > (ULLONG_MAX - tmp_n) / carry)
                            overflow_flag = true;
                     tmp_n += carry * tmp_dn;
                     carry = (unsigned long)(tmp_n / (2 * tmp_dn));
```

```
                      acc_n[j] = tmp_n % (tmp_dn * 2);
                      acc_dn[j] = tmp_dn;
                      }

              if (first_time == true)
                      {
                      tmp_n = f / 2;
                      acc_n[0] = (tmp_n + carry);
                      acc_dn[0] = 1;
                      dig_n = (unsigned)(acc_n[0] / f);
                      }
              else
                      dig_n = (unsigned)(acc_n[0] + carry / f);
              car = (unsigned)(dig_n / f);
              dig_n %= f;
              // Add the carry to the existing number for ln(2) calculate so far.
              if (car > 0)
                      {
                      ++no_carry;
                      for (int j = ss.length(); car > 0 && j > 0; --j)
                              {
                              int dd;
                              dd = (ss[j - 1] - '0') + car;
                              car = dd / 10;
                              ss[j - 1] = dd % 10 + '0';
                              }
                      }
              (void)sprintf(buffer, "%0*lu", first_time == true ? 1 : no_dig,
dig_n);
              ss += std::string(buffer);
              if (first_time == true)
                      acc_n[0] %= f;
              else
                      acc_n[0] = carry % f;
              carry = 0; // carry %= f;
              }

       ss.insert(1, ".");// add a come after the first digit to create 0.69...
       if (overflow_flag == false)
              ss.erase(digits + 1); // Remove the extra digits that we didnt
requested.
       else
              ss = std::string("Overflow:") + ss;

       delete acc_n;
       delete acc_dn;
       return ss;
       }
```

The above mention algorithm can find ln(2) and for each loop, we can find between one and eight digits at a time. Not surprisingly, the more digits we find per loop the faster the overall algorithm is as shown in below diagram. The number 1 to 8 refer to how many digits we find per loop and in calculation, ln(2) with digits from 32 to 32,768 digits and the timing on the Y-axis is milliseconds. For reference, also the Taylor series expansion of ln(2) using arbitrary precision is also show (LN2 table). It is quite interesting to see that the spigot algorithm for ln(2) beat the traditional way of calculation ln(2) using arbitrary precision with a factor of approximately 80 times compare to the spigot algorithm finding ln(2) with 8 digits at a time.



And in BBP style notation: $\frac{1}{2}P(1,2,1,(1))$

Spigot algorithm for ln(10)

Next in turn is ln(10) another useful transcendental constant. The series expansion for ln(x) is $\sum_{n=1}^{\infty}\frac{1}{n}(\frac{x-1}{x})^n$. For x=10 you get ln(10)= $\sum_{n=1}^{\infty}\frac{1}{n}(\frac{9}{10})^n$ as usually we need to rewite the series expansion into a horner type representation and you get:

$$\ln(10)=\sum_{n=1}^{\infty}\frac{1}{n}(\frac{9}{10})^n =>$$

$$\ln(10)=\frac{9}{10}+\frac{1}{2}(\frac{9}{10})^2+\frac{1}{3}(\frac{9}{10})^3+\frac{1}{4}(\frac{9}{10})^4+\frac{1}{5}(\frac{9}{10})^5+\cdots =>$$

$$\ln(10)=\left(\frac{9}{10}+\frac{9}{10}\left(\frac{1}{2}\frac{9}{10}+\frac{1}{3}(\frac{9}{10})^2+\frac{1}{4}(\frac{9}{10})^3+\frac{1}{5}(\frac{9}{10})^4+\cdots\right)\right) =>$$

$$\ln(10)=\left(\frac{9}{10}+\frac{9}{10}\left(\frac{9}{20}+\frac{9}{10}\left(\frac{1}{3}\frac{9}{10}+\frac{1}{4}(\frac{9}{10})^2+\frac{1}{5}(\frac{9}{10})^3+\cdots\right)\right)\right) =>$$

$$\ln(10)=\left(\frac{9}{10}+\frac{9}{10}\left(\frac{9}{20}+\frac{9}{10}\left(\frac{9}{30}+\frac{9}{10}\left(\frac{1}{4}\frac{9}{10}+\frac{1}{5}(\frac{9}{10})^2+\cdots\right)\right)\right)\right) =>$$

$$\ln(10)=\left(\frac{9}{10}+\frac{9}{10}\left(\frac{9}{20}+\frac{9}{10}\left(\frac{9}{30}+\frac{9}{10}\left(\frac{9}{40}+\frac{9}{10}\left(\frac{1}{5}\frac{9}{10}+\cdots\right)\right)\right)\right)\right) =>$$

$$\ln(10)=(\frac{9}{10}+\frac{9}{10}\left(\frac{9}{20}+\frac{9}{10}(\frac{9}{30}+\frac{9}{10}(\frac{9}{40}+\frac{9}{10}\left(\frac{9}{50}+\cdots\right)))\right))$$

This is the way we want to have the series expanded so we can quickly identifies the different Spigot elements. This is a mixed-radix base $c=\left(\frac{9}{10},\frac{9}{20},\frac{9}{30},\frac{9}{40},\frac{9}{10},\dots\right)$ with respect to ln(10)=($\frac{9}{10};\frac{9}{10},\frac{9}{10},\frac{9}{10}$, ...).

Now we need to figure out how many terms we would need for a giving number of wanted digits, *d*. We are a little bit worry since the factor for each term is multiply with 0.9 which is nearly twice as high as the 0.5 for ln(2). Therefore, we would expect a much slower convergence rate that translate into more terms is needed for a giving number of wanted digits *d*.

If we see the fraction between two terms in the series expansion we get $\frac{\frac{1}{(n+1)}(\frac{9}{10})^{n+1}}{\frac{1}{n}(\frac{9}{10})^{n}}=\frac{n(\frac{9}{10})}{(n+1)}=\frac{9}{10(1+\frac{1}{n})}$ for large n it is ~ 0.9. For ln(2) we got ~ 0.5 for each term, so for ln(10) we need $0.9^n=0.5$ more terms than for ln(2). Taken ln() on both side we get n=$\frac{\ln(0.5)}{\ln(0.9)}$~6.6 more terms.

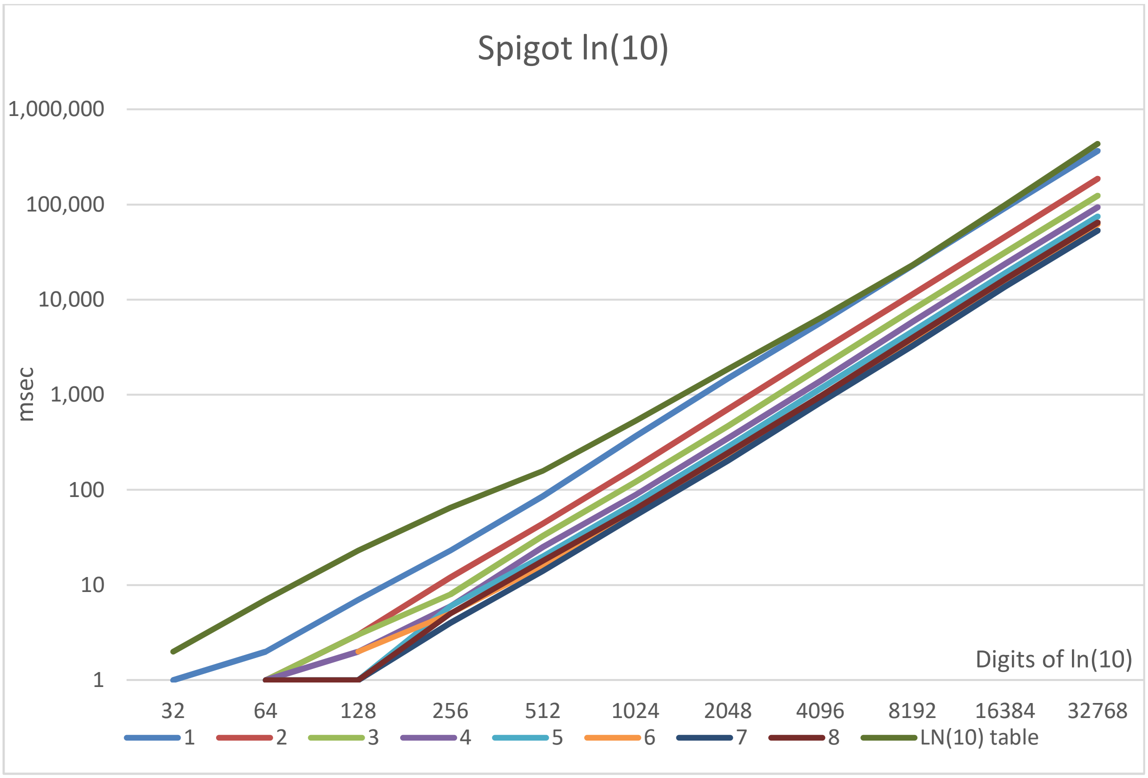
Algorithm 5.1 spigot_ln_64()

```
// 64 bit version of spigot algorithm for LN(10)
// It has automatic 64bit integer overflow detection in which case the result
start with the string "Overflow...."
// A Column: x-1,x-1,x-1,...,x-1
// B Column: x,x,x,x,x,...,x
// Initialization values: (x-1)/(x(n+1))...
std::string spigot_ln_64( unsigned int x, int digits, int no_dig = 1)
	{
	static unsigned long f_table[] = { 1, 10, 100, 1000, 10000, 100000,
1000000, 10000000, 100000000 };
	bool first_time = true;			// First iteration of the algorithm
	bool overflow_flag = false;		// 64bit integer overflow flag
	bool bit32_overflow = false;
	char buffer[32];
	std::string ss;				// The std::string that holds the ln(x)
	int dig;
	unsigned int car, no_carry = 0;
	unsigned int no_terms;		// No of terms to complete as a function of
digits
	unsigned long f;			// New base 1 decimal digits at a time
	unsigned long dig_n;		// dig_n holds the next no_dig digit to add
	unsigned _int64 carry;
	unsigned _int64 tmp_n, tmp_dn;
	ss.reserve(digits + 16);
	int factor;

	if (x < 1) return std::string("Domain Error of argument. Required x>=1");

	if (no_dig > 8) no_dig = 8;		// ensure no_dig<=8
```

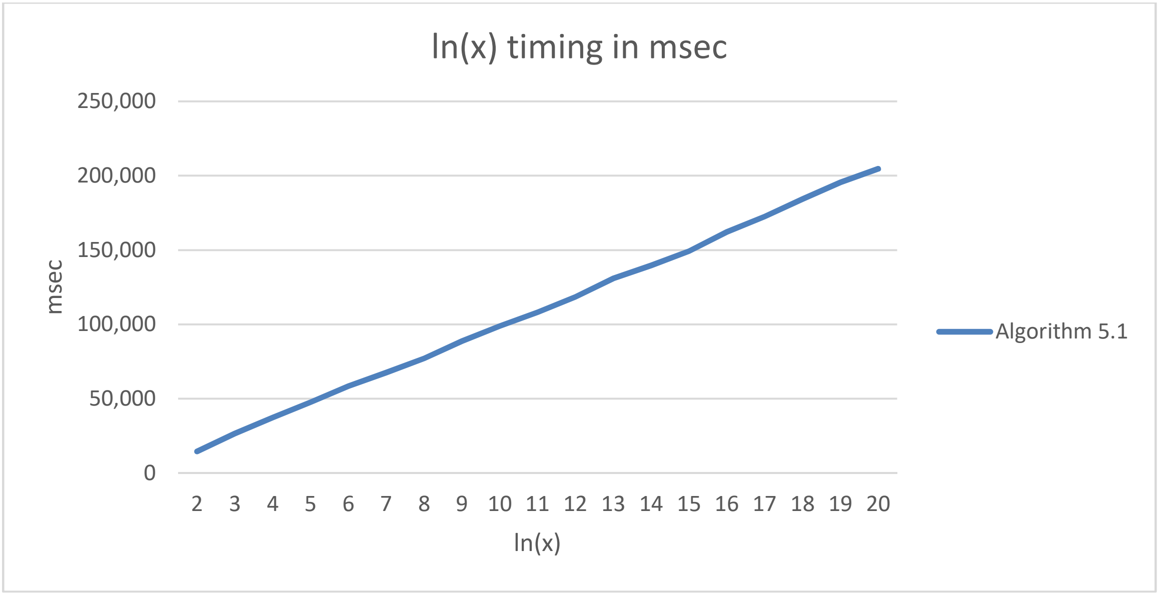
```
       if (no_dig < 1) no_dig = 1;        // Ensure no_dig>0
       // Since we do it in trunks of no_dig digits at a time we need to ensure
digits is divisble with no_dig.
       dig = (digits / no_dig + (digits%no_dig>0 ? 1 : 0)) * no_dig;
       dig += no_dig;                     // Extra guard digits
       // Calculate the number of terms needed
       factor=(int)ceil(10*log(0.5) / log((double)(x - 1) / (double)x));
       no_terms = (unsigned int)(factor * dig / 3 + 3);

       // Allocate the needed accumulators
       unsigned _int64 *acc_n = new unsigned _int64[no_terms + 1];
       unsigned _int64 *acc_dn = new unsigned _int64[no_terms + 1];
       f = f_table[no_dig];        // Load the initial f
       carry = 0;                  // Set carry to 0
       //Loop for each no_dig
       for (int i = dig; i >= 0 && overflow_flag == false; i -= first_time == true
? 1 : no_dig, first_time = false)
              {
              // Calculate new number of terms needed
              no_terms = (unsigned int)(factor * i / 3 + 3);
              // Loop for each no_terms
              for (int j = no_terms; j>0 && overflow_flag == false; --j)
                     {
                     if (first_time == true)
                            {// Calculate the initialize value
                            tmp_dn = (j + 1) * x;
                            tmp_n = (x-1);
                            }
                     else
                            {
                            tmp_n = acc_n[j];
                            tmp_dn = acc_dn[j];
                            }
                     if ( tmp_n > (ULLONG_MAX)/f )
                            overflow_flag = true;
                     tmp_n *= f;           // Scale it
                     // Check for 64bit overflow. Not very likely
                     if (carry > 0 && tmp_dn > (ULLONG_MAX - tmp_n) / carry)
                            overflow_flag = true;
                     tmp_n += carry * tmp_dn;
                     carry = (tmp_n / (x * tmp_dn));
                     carry *= (x-1);
                     acc_n[j] = tmp_n % (tmp_dn * x);
                     acc_dn[j] = tmp_dn;
                     }

              if (first_time == true)
                     {
                     tmp_n = (x-1) * f;
                     if (carry > 0 && tmp_n > (ULLONG_MAX - carry * x ))
                            overflow_flag = true;
                     acc_n[0] = (tmp_n + carry*x);
                     acc_dn[0] = x;
                     dig_n = (unsigned)(acc_n[0] / (f*acc_dn[0]));
                     }
              else
                     {
                     if (acc_n[0] > (ULLONG_MAX - carry * acc_dn[0]) / f)
```

```
                            overflow_flag = true;
                     dig_n = (unsigned)((acc_n[0] * f + carry * acc_dn[0]) /
(f*acc_dn[0]));
                     }
              car = (unsigned)(dig_n / f);
              dig_n %= f;
              // Add the carry to the existing number for ln(x) calculate so far.
              if (car > 0)
                     {
                     ++no_carry;
                     for (int j = ss.length(); car > 0 && j > 0; --j)
                            {
                            int dd;
                            dd = (ss[j - 1] - '0') + car;
                            car = dd / 10;
                            ss[j - 1] = dd % 10 + '0';
                            }
                     }
              (void)sprintf(buffer, "%0*lu", first_time == true ? 1 : no_dig,
dig_n);
              ss += std::string(buffer);
              if (first_time == true)
                     acc_n[0] %= f*acc_dn[0];
              else
                     {
                     acc_n[0] = acc_n[0] * f + carry *acc_dn[0];
                     acc_n[0] %= f  * acc_dn[0];
                     }
              carry = 0;
              }

       ss.insert(1, ".");// add a come after the first digit to create 0.69...
       if (overflow_flag == false)
              ss.erase(digits + 1); // Remove the extra digits that we didnt
requested.
       else
              ss = std::string("Overflow:") + ss;

       delete acc_n, acc_dn;
       return ss;
       }
```

The above mention algorithm can find ln(10) and for each loop, we can find between one and eight digits at a time. Not surprisingly, the more digits we find per loop the faster the overall algorithm is as shown in below diagram. The number 1 to 8 refer to how many digits we find per loop and in calculation, ln(10) with digits from 32 to 32,768 digits and the timing on the Y-axis is milliseconds. For reference, also the Taylor series expansion of ln(10) using arbitrary precision is also shown (LN10 table). It is quite interesting to see that the spigot algorithm for ln(10) beat the traditional way of calculation ln(10) using arbitrary precision with a factor of approximately 8 times compare to the spigot algorithm finding ln(10) 7 digits at a time. This is much less than the speedup for ln(2) versus the traditional algorithm using arbitrary precision. The reason is that the arbitrary precision algorithm are using argument reduction to speed up the Taylor series for ln(10) a technique we unfortunately can't use in in the current implementation of the spigot algorithm and therefore we will see that the speed advantages using a spigot algorithm will diminish with higher number of ln(x).



Algorithm 5.1 can in principle handle x>1. However as we increase x we also notice that the convergence rate decrease requiring more terms per digits. Below graph, show the timing of the algorithm for various x calculating four digits at a time. As can we seen we have a linear dependency of x and as x increase the advantaged of using this spigot algorithm diminish. Around x>50 the algorithm performed worse than the arbitrary precision version.



I mention before that we unfortunately cannot apply the speed up trick of argument reduction. However, that is not entirely correct. If we look at ln(10) we could also rewrite it to ln(10)=ln(2*5)=ln(2)+ln(5). This at first glance seems counterproductive since we would now have to call our Algorithm 5.1 twice. First with ln(2) and secondly with ln(5) and then have to add them together. However adding two arbitrary precision number is an O(n) complexity and therefore fast so it will not contribute to the overall calculation time. Based on the above graph for performance ln(2) can be done in 14.5sec for 32,768 digits and ln(5) 47.7sec. Adding them together, we get 62.2sec compared to the 99sec for doing ln(10) directly and 37% speed improvement. Great however we can do better. Continue along the above previous line we could also rewrite ln(10) as:

$$\ln(10) = \ln\left(2^3\frac{10}{2^3}\right) = \ln(2^3) + \ln\left(\frac{10}{8}\right) = 3\ln(2) + \ln(\frac{10}{8})$$

Now both argument is small ln(2) and ln(1.25) respectively and we expect than ln(1.25) can be done faster since it is smaller than ln(2). A multiplying an arbitrary precision number with a single digit constant 3 is also of O(n) complexity and therefore fast.
Now we only need to change algorithm 5.1 to be able to be called with a fraction $(\frac{10}{8})$.
However, that is surprisingly easy since the algorithm itself is working on fraction. The only think we need is to change is the initialize value to a fraction instead of an integer. See Algorithm 5.2 below.

Algorithm 5.2 Spigo_lnxy_64()

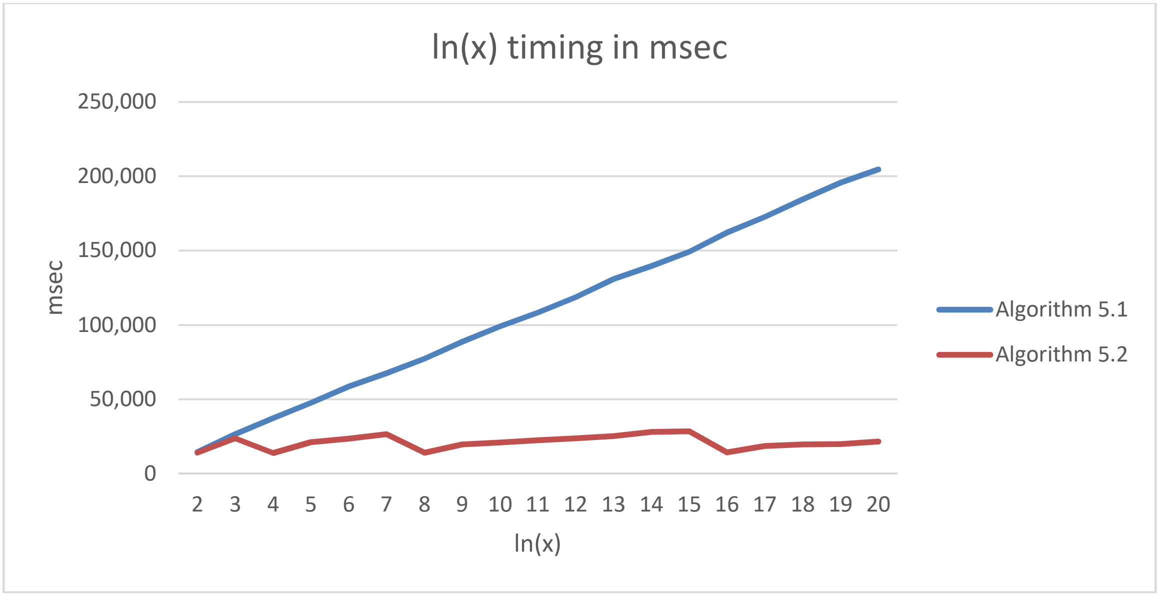
```
// 64 bit version of spigot algorithm for LN(x/y) fraction
// It has automatic 64bit integer overflow detection in whcih case the result
start with the string "Overflow...."
// A Column: x-1,x-1,x-1,...,x-1
// B Column: x,x,x,x,x,...,x
// Initialization values: (x-1)/(x(n+1))...
std::string spigot_lnxy_64(unsigned int x, unsigned int y, int digits, int no_dig
= 1)
       {
```

```
       static unsigned long f_table[] = { 1, 10, 100, 1000, 10000, 100000, 
1000000, 10000000, 100000000 };
       bool first_time = true;            // First iteration of the algorithm
       bool overflow_flag = false;        // 64bit integer overflow flag
       bool bit32_overflow = false;
       char buffer[32];
       std::string ss;                    // The std::string that holds the ln(x)
       int dig;
       unsigned int car, no_carry = 0;
       unsigned int no_terms;             // No of terms to complete as a function 
of digits
       unsigned long f;                   // New base 1 decimal digits at a time
       unsigned long dig_n;               // dig_n holds the next no_dig digit to 
add
       unsigned _int64 carry;
       unsigned _int64 tmp_n, tmp_dn;
       ss.reserve(digits + 16);
       int factor;

       if (x < y) return std::string("Domain Error of argument.Required x>y");
       if (x <= 0) return std::string("Domain Error of argument. Required x>0");

       if (no_dig > 8) no_dig = 8;               // ensure no_dig<=8
       if (no_dig < 1) no_dig = 1;               // Ensure no_dig>0
       // Since we do it in trunks of no_dig digits at a time we need to ensure 
digits is divisble with no_dig.
       dig = (digits / no_dig + (digits%no_dig>0 ? 1 : 0)) * no_dig;
       dig += no_dig;                                          // Extra guard 
digits
       // Calculate the number of terms needed
       factor = (int)ceil(10 * log(0.5) / log((double)(x - y) / (double)x));
       no_terms = (unsigned int)(factor * dig / 3 + 3);
       // Allocate the needed accumulators
       unsigned _int64 *acc_n = new unsigned _int64[no_terms + 1];
       unsigned _int64 *acc_dn = new unsigned _int64[no_terms + 1];
       f = f_table[no_dig];        // Load the initial f
       carry = 0;                  // Set carry to 0
       //Loop for each no_dig
       for (int i = dig; i >= 0 && overflow_flag == false; i -= first_time == true 
? 1 : no_dig, first_time = false)
              {
              // Calculate new number of terms needed
              no_terms = (unsigned int)(factor * i / 3 + 3);
              // Loop for each no_terms
              for (int j = no_terms; j>0 && overflow_flag == false; --j)
                     {
                     if (first_time == true)
                            {// Calculate the initialize value
                            tmp_dn = (j + 1) * x;
                            tmp_n = (x - y);
                            }
                     else
                            {
                            tmp_n = acc_n[j];
                            tmp_dn = acc_dn[j];
                            }
                     if (tmp_n > (ULLONG_MAX) / f)
                            overflow_flag = true;
```

```
                    tmp_n *= f;          // Scale it
                    // Check for 64bit overflow. Not very likely
                    if (carry > 0 && tmp_dn > (ULLONG_MAX - tmp_n) / carry)
                           overflow_flag = true;
                    tmp_n += carry * tmp_dn;
                    carry = (tmp_n / (x * tmp_dn));
                    carry *= (x - y);
                    acc_n[j] = tmp_n % (tmp_dn * x);
                    acc_dn[j] = tmp_dn;
                    }

             if (first_time == true)
                    {
                    tmp_n = (x - y) * f;
                    if (carry > 0 && tmp_n > (ULLONG_MAX - carry * x))
                           overflow_flag = true;

                    acc_n[0] = (tmp_n + carry*x);
                    acc_dn[0] = x;
                    dig_n = (unsigned)(acc_n[0] / (f*acc_dn[0]));
                    }
             else
                    {
                    if (acc_n[0] > (ULLONG_MAX - carry * acc_dn[0]) / f)
                           overflow_flag = true;
                    dig_n = (unsigned)((acc_n[0] * f + carry * acc_dn[0]) /
(f*acc_dn[0]));
                    }
             car = (unsigned)(dig_n / f);
             dig_n %= f;
             // Add the carry to the existing number for ln(x/y) calculated.
             if (car > 0)
                    {
                    ++no_carry;
                    for (int j = ss.length(); car > 0 && j > 0; --j)
                           {
                           int dd;
                           dd = (ss[j - 1] - '0') + car;
                           car = dd / 10;
                           ss[j - 1] = dd % 10 + '0';
                           }
                    }
             (void)sprintf(buffer, "%0*lu", first_time == true ? 1 : no_dig,
dig_n);
             ss += std::string(buffer);
             if (first_time == true)
                    acc_n[0] %= f*acc_dn[0];
             else
                    {
                    acc_n[0] = acc_n[0] * f + carry *acc_dn[0];
                    acc_n[0] %= f  * acc_dn[0];
                    }
             carry = 0;
             }

      ss.insert(1, ".");// add a come after the first digit to create 2.30...
      if (overflow_flag == false)
```

```
		ss.erase(digits + 1); // Remove the extra digits that we didnt 
requested.
	else
		ss = std::string("Overflow:") + ss;

	delete acc_n, acc_dn;
	return ss;
	}
```

Running the same performance chart with Algorithm 5.2 and ln(x) from 2 to 20 we get the following performance graph for comparison.



Algorithm 5.2 totally outperformed algorithm 5.1. The reason for the jagged line of Algorithm 5.2 is that the fraction various between one and two dependently on the ln(x) and that affect the performance. However, it scale much better than algorithm 5.1 with a significant performance gain as ln(x) increases. As an example we could rewrite ln(18) as:

$$\ln(18) = \ln\left(2^4 \frac{18}{2^4}\right) = \ln(2^4) + \ln\left(\frac{18}{16}\right) = 4\ln(2) + \ln(\frac{18}{16})$$

Instead of 3, we now multiply with 4, which does not cost of any performance loss and the fraction $\frac{18}{16} = 1.125$ is very close to 1 and therefore fast. Notice also the dip for each number that is a power of two. E.g. 2, 4, 8 and 16 for these numbers the addition element will also be ln(1) and therefore zero so there are nothing to add. The worst case scenario will always be when the number is 2^n-1.In these case the adding element becomes $\ln(\frac{2^n-1}{2^n}) \approx \ln(2)$ in other words the worst case timing is 2 times the time for the ln(2) calculation regardless of the number x. The means that the algorithm scales very well even for very higher number of x.

When comparing the new algorithm 5.2 with the traditional way of calculating ln(10) using arbitrary previous we see that the algorithm is more than 20-40 times faster than the traditional calculation.

Unbounded Spigot algorithm for π

All the previous spigot algorithm requires us to know in advance the number of digits we want. However, Gibbons [13] outline a way to compute in a steady stream the digits of π without prior knowledge of how many digit we need. The below source produce a steady stream of π digits using the author own Arbitrary precision library. The source is a port from another source from which I have lost the reference.

Algorithm 6.1 unbounded_pi()

```
// Unbounded PI algorithm
void unbounded_pi()
       {
       const int_precision c1(1), c4(4), c7(7), c10(10), c3(3), c2(2);
       int_precision q(1), r(0), t(1);
       unsigned k = 1, l = 3, n = 3;
       int_precision nn, nr;
       int i, j;

       for (i = 0, j = 0; ; ++j)
              {
              if ((c4*q + r - t) < n*t)
                     {
                     cout << (char)(n + '0') << flush;
                     i++;
                     if (i == 1)
                            cout << "." << flush;
                     nr = c10*(r - (n*t));
                     n = (int)((c3*q + r) / t) - n;
                     q *= c10;
                     r = nr;
                     }
              else {
                     nr = (c2*q + r)*int_precision(l);
                     q *= k;
                     t *= l;
                     nn = (q*c7 + c2 + r*l) / t;
                     l += 2;
                     k += 1;
                     n = nn;
                     r = nr;
                     }
              }
       }
```

Reference

1. The World of π. www.pi314.net/eng/salamin.php - Oct 5 2016
2. A. H. J. Sale, The Calculation of e to many significant digits, http://comjnl.oxfordjpurmals.org
3. Borwein, Pi and the AGM, Volume 4, John Willey & Sons Inc, New York, NY 1998
4. https://en.wikipedia.org/wiki/Borwein%27s_algorithm – Oct 6 2016
5. https://en.wikipedia.org/wiki/Bailey%E2%80%93Borwein%E2%80%93Plouffe_formula – Oct 6, 2016
6. https://en.wikipedia.org/wiki/Approximations_of_%CF%80 – Oct 6, 2016
7. P. Borwein – The Amazing number π
8. Bailey, Borwein, Plouffe, The Quest for Pi, June 25 1996
9. J. Borwein, Ramanujan and PI, May 3 2012
10. J Borwein, The life of Pi: From Archimedes to Eniac and Beyond, June 19, 2012
11. Bailey, Borwein, Plouffe, "on the rapid Computation of Various Polylogarithmic Constants 1996. http://www.cecm.sfu/personal/pborwein
12. Rabinowitz & Wagon, A Spigot Algorithm for the Digits of Pi, The American Mathematical Monthly, 102 (1995) page 195-203.
13. Unbounded Spigot Algorithms for the Digits of PI
14. The world of π. http://www.pi314.net/eng/goutte.php - Dec 28 2016
15. D Bailey, A compendium of BBP-Type Formulas for Mathematical Constants. April 29, 2013