Practical implementation of π Algorithms

Practical implementation of π Algorithms.

By Henrik Vestermark (hve@hvks.com)

Abstract:

This paper examined the various modern version of algorithms for calculating π. That potential could be based on using these algorithms when using arbitrary precision arithmetic that opens up for calculating π to Billions or Trillions of digits. Let it be noted that there is no engineering or practical reason why you need to calculate π beyond the limitation in the IEEE754 standard for floating point precision as found in PC, computers, etc. However, this quest for more precision has led to a lot of discovery of modern and faster algorithm that is presented in this paper. This revision has been updated with a new section on the Binary Splitting method. Currently, the record in 2022 is 100 trillions digits of π.

Introduction:

We have more or less always been used to having the constant π readily available in our handheld calculator, Excel sheet, or as a hardware instruction in the CPU.
However usually only available at the native precision of the Calculator or the IEEE754 standard that gives access to the 64-bit floating-point constant of π with approx. 15 digits accuracy.
However, if we want to venture outside this limited range of precision we are on our own, meaning we have to resort to any of the many available formulas for finding π with higher precision.

In this paper, we look at the various methods available to us and as usual, we look into the practical implementation of calculating π at a higher precision than what is available with the IEEE 754 floating point standard.

Before we do that, we will first establish a number of the popular algorithm for finding π just using the standard IEE754 floating-point calculation as an example and then later on show the result when these algorithms are applied using arbitrary precision arithmetic.

The paper is divided into four sections. Section 1 is calculating π using a classic newton iteration, and in section 2 we are looking into the infinite series by Ramanujan, Chudnovsky, and the Borwein brothers. In section 3, we looked at higher order iteration to calculate π, which has been dominated by the Borwein brothers and their research and finally in section 4 we introduce the bounded spigot algorithm as an alternative way of generating π, which many times does not require us to resort to arbitrary precision arithmetic. As always, we list c++ source code for the practical implementation of these algorithms.

Change log

For version February 7, 2023. Correcting grammars.
For version January 17, 2023. We have added the relative cost of the binary splitting method in the Appendix.
For version August 2022. We have:

- Expanded the general introduction about the Author's own arbitrary precision library packages.
- In general, all performance charts have been updated to reflect the newer binary version of the author's arbitrary precision libraries.
- Added the binary splitting method for the Ramanujan, Chudnovsky, Borwein25, and Borwein50 infinite series.
- Added the Borwein 1993 Infinite series producing approx. 50 digits per iteration.
- The Gosper spigot algorithm has been added.
- Added an Appendix where the Binary splitting method is derived for Ramanujan, Chudnovsky, and Borwein

Practical implementation of π Algorithms

Contents

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 exponent (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

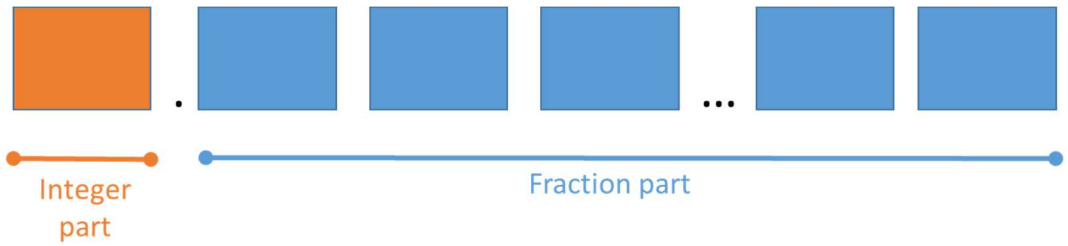
For the internal layout of the arbitrary precision number, we are using the STL vector library declared as:

```
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].

Newton's method for calculating π

Imagine that we do not have access to π at all or at least not with the required precision, we would need to calculate π, typically through some sort of iteration until the desired precision is obtained.

A simple, yet efficient method is finding π through the equation:

$$\sin(x) = 0$$

Which has the general solution of $x = n\pi$; we are of course interested in finding the solution for n=1 and applying Newton's iteration formula of:

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

We get:

$$x_{n+1} = x_n + \frac{sin(x_n)}{cos(x_n)} \quad \text{Or} \quad x_{n+1} = x_n + \tan(x_n)$$

To see how it works in real life we can e.g. start with x_0=2 and get the following iteration.

Iteration	π	F(x)=sin(x)	F'(X)=cos(x)	Error
0	2.00	9.09E-01	-4.16E-01	1.14E+00
1	4.18503986326152	-8.64E-01	-5.03E-01	1.04E+00
2	2.46789367451467	6.24E-01	-7.82E-01	6.74E-01
3	3.26618627756911	-1.24E-01	-9.92E-01	1.25E-01
4	3.14094391231764	6.49E-04	-1.00E+00	6.49E-04
5	3.14159265368080	-9.10E-11	-1.00E+00	9.10E-11
6	3.14159265358979	1.23E-16	-1.00E+00	0.00E+00

We see quick convergence and after six iterations, we have the solution to the accuracy of approx. 15 digits. The Newton method is said to have quadratic convergence meaning that the number of correct digits doubles for each iteration.

We also notice that

$$\lim_{x \to \pi} \sin(x) \to 0$$

And

$$\lim_{x\to\pi} \cos(x) \to 1$$

So we can simplify the iteration by simply ignoring cos(x) and iterate using:

$$x_{n+1} = x_n + \sin(x_n)$$

Iteration	π	Sin(x)	Error
0	2.00	0.909297	1.14E+00
1	2.90929742682568	0.230212	2.32E-01
2	3.13950913306779	0.002084	2.08E-03
3	3.14159265208235	1.51E-09	1.51E-09
4	3.14159265358979	1.23E-16	0.00E+00

After four iterations, we get the same accuracy as in the normal case. The fact that it finishes with the same accuracy after four instead of the original six iterations is not a sign that the new formula is faster. It still has quadratic convergence like the original doubling the number of accurate digits in each iteration. However, we avoided the calculation of cos(x) making the algorithm twice as fast as the original. This is very important when using arbitrary precision arithmetic. The trigonometric function is very expensive time-wise to calculate.

Can we further improve on the newton formula? The answer is yes. Instead of iterating where the new x_{n+1} is the tangent interception on the x of the previous point x_n, we can use a polynomial curve of higher order. One use is the 9th-order iteration

$$x_{n+1} = x_n + \sin(x_n) + \frac{1}{6}sin^3(x_n) + \frac{3}{40}sin^5(x_n) + \frac{5}{112}sin^7(x_n)$$

Which has a much faster convergence, giving 9 times as many correct digits for each iteration.

Iteration	π	Sin(x)+6Sin(x)^3/40+3Sin(x)^5/40+5Sin(x)^7/112	Error
0	2	1.104169235	1.14E+00
1	3.10416923500490	0.037423419	3.74E-02
2	3.14159265358979	4.56341E-15	4.44E-15
3	3.14159265358979	1.22515E-16	0.00E+00

After only three iterations, we have found the solution with 15 digits accuracy.
Now is it worth using these higher-order iterations? Assuming the most time-consuming calculation is sin(x) which is several magnitudes higher than regular arithmetic the answer is yes since we in both places calculate $\sin(x_n)$ once. Of course, we will also need to revise the nine-order method by rearranging it a little bit:

$$x_{n+1} = x_n + \frac{1}{1680}\sin(x_n)(1680 + sin^2(x_n)(280 + sin^2(x_n)(126 + 75sin^2(x_n))))$$

The constant $\frac{1}{1680}$ can be calculated before the iteration and changed into a multiplication, which is a much faster operation in arbitrary precision arithmetic.

The two Newton iteration code segments are listed below.

Newton (2^{nd} order convergence)

Notice that we have "seed" the iteration with the first approx. 15-16 digits of π effectively avoiding 4 iterations steps. E.g. if we want π with 262,144 digits (2^{18}) but make our initial start guess with the first 16 digits (2^4) of π we only need to iterate through 18-4=14 iterations to get our result. For a large number of digits that will save us a tremendous amount of calculation, equivalent to approx. 22% time savings.

Algorithm 1.1 Standard Newton

```
// Standard newton iteration of PI given by x=x+sin(x)
float_precision pi_newton(const uintmax_t digits)
	{
	size_t loopcnt=0;
	// Set default 16 digits precision as a starting point.
	float_precision piprev(0,digits+1), pi("3.141592653589793", digits+1);

	if( digits <= 16)  // Do we already have the digits?
		return pi;

	for (; pi!=piprev; ++loopcnt )
		{
		piprev = pi;
		pi = piprev + sin(piprev);
		}

	return pi;
	}
```

Newton(9^{th} order convergence)

9^{th} order requires fewer iterations from the example above it only requires 9^x=262,144 or approx. 5.7 iterations, while starting with the first 16 correct digits of π gives a net effect of approx. 4.4 iterations again a saving of around 23%.

Algorithm 1.2 Newton's ninth-order convergence method

```
// 9th order newton iteration for PI using
// x=x+sin(x)+1/6Sin(x)^3+3/40sin(x)^5+5/112sin(x)^7
float_precision pi_newton9(const uintmax_t digits)
	{
	const size_t precision=digits + 1;
	size_t loopcnt = 0;
	const float_precision c1680(1680),c280(280),c126(126),c75(75); // constants
```

```
	// Set default 16 digits precisionas a starting point.
	float_precision pi("3.141592653589793", precision);
	float_precision sinx(0,precision), sinxsq(0,precision),
			inv1680(1680,precision), r(0,precision); //temp variables

	if (digits <= 16)  // Do we already have the digits?
		return pi;

	inv1680 = inv1680.inverse();
	for (; ; ++loopcnt)
		{
		sinx = sin(pi);
		sinxsq = sinx.square();
		r = inv1680*(sinx*(c1680+sinxsq*(c280+sinxsq*(c126+c75*sinxsq))));
		if (pi + r == pi)
			break;
		pi += r;
		}
	return pi;
	}
```

Improvement of the Newton method?

Further improvements can be made by noting that a Newton iteration step is self-correcting. In the previous example of π with 262,144 digits, instead of starting using arbitrary precision with 262,144 digits precision, we could start with a smaller precision and then progressively increase the precision until we in the last iteration reach the desired goal of precisions. The author calls this Newton method with dynamic precision or iterative deepening (increasing precision per iteration)

The two algorithms of progressively more digits to calculate π is listed below.

Algorithm 1.3 Newton Standard with Dynamic Precision

```
// Newton standard with iterative deepening
float_precision pi_newton_deepening(const uintmax_t digits)
	{
	uintmax_t d;
	size_t loopcnt = 0;
	// Set default 16 digits precision as a starting point.
	float_precision piprev(0, digits + 1), pi("3.141592653589793", digits + 1);

	if (digits <= 16)  // Do we already have the digits?
		return pi;

	for (d = std::min(digits, (uintmax_t)32); pi != piprev; ++loopcnt )
		{
		piprev.precision(d+1);   // change the calculating precision
		pi.precision(d+1);

		piprev = pi;
		pi = piprev + sin(piprev);
		// Increase the calculating precision by the convergence rate of 2
		d = std::min(digits, 2 * d );
```

```
            }

      loopcnt_newton = loopcnt;
      return pi;
      }
```

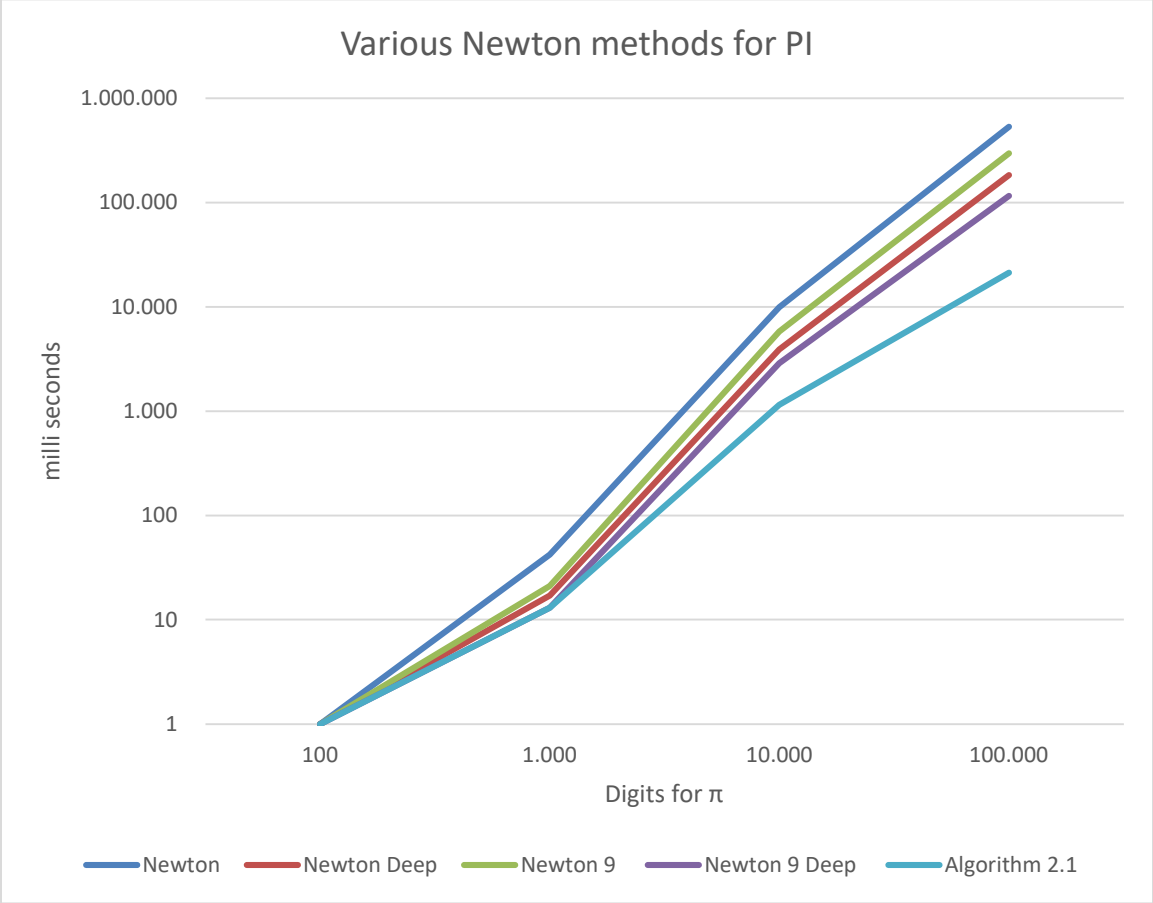
Algorithm 1.4 Newton ninth-order convergence using dynamic precision

```
// Newton 9'th order with iterative deepening
float_precision pi_newton9_deepening(const uintmax_t digits)
      {
      uintmax_t d;
      size_t loopcnt = 0;
      const size_t precision = digits + 1;
      const float_precision c1680(1680),c280(280),c126(126),c75(75); // constants
      // Set default 15 digits precision as a starting point.
      float_precision pi("3.141592653589793", precision);
      float_precision sinx(0,precision), sinxsq(0, precision),
            inv1680(1680, precision), r(0,precision); //temp variables

      if (digits <= 16)  // Do we already have the digits?
            return pi;

      inv1680 = inv1680.inverse();
      for (d = std::min(digits, (uintmax_t)(16*9)); ;++loopcnt )
            {
            // change the calculating precision
            pi.precision(d + 1);
            sinx.precision(d + 1);
            sinxsq.precision(d + 1);
            r.precision(d + 1);

            sinx = sin(pi);
            sinxsq = sinx.square();
            r=inv1680*(sinx*(c1680+sinxsq*(c280+sinxsq*(c126+c75*sinxsq))));
            if (pi + r == pi)
                  break;
            pi += r;
            // Increase the calculating precision by the convergence rate of 9
            d = std::min(digits, 9 * d);
            }

      return pi;
      }
```

The performance improves significantly by doing iterative deepening instead of the regular iteration, however, it still doesn't match up to our standard algorithm in the arbitrary precision package we are using, particular when increasing the number of π digits. The Newton algorithm for dynamic precision is as expected a huge improvement over the standard Newton methods and as an example, it speeds up the Newton algorithm with a factor of approx. three over the standard Newton method and a factor of over 2.5 in the 9^{th}-order Newton convergence method using dynamic precision. The ninth-order Newton method is approx. 1.5 times faster than the Newton 2^{nd} order method.

However, in the end, the Newton method won't have the performance of e.g. the Borwein Algorithm 2.1 which shows much better scalability when the number of digits goes up. The culprit is the slow calculation of sin(x) versus Borwein's algorithm 2.1 which only use the basic arithmetic function, like +, -, *, / and √.

See the below chart with a comparison to Borwein algorithm 2.1 describe in section 3 of this paper.



Recommendation for Newton's methods for generating π

The following recommendation arises for examining the Newton methods.

- The Newton method is not the faster way of generating π and is therefore not recommended
- If you want to use a Newton method then use the ninth-order Newton method using dynamic precision.

Infinite series for π

Practical implementation of π Algorithms

In this section, we examine the famous Ramanujan, Chudnovsky, and Borwein infinite series for finding π. The series' performance in itself is not impressive compared to other methods. However, you can apply the binary splitting method that significantly improved the performance of the infinite series.

Ramanujan and π

Ramanujan was an Indian self-study mathematician who around 1910, quite astonished, invented the following infinite series formula for π.

$$\frac{1}{\pi} = \frac{2\sqrt{2}}{9801}\sum_{n=0}^{\infty}\frac{(4n)!\,(1103+26390n)}{(n!)^4 396^{4n}}$$

For each term in the series, it produces approx. eight more correct digits of π. In the quest for calculating π, this particular formula was used in 1985 to calculate approx. 17 million digits of π. In the table below, we use column a_n to denote the n^{th} term of the summations and the column ∑ to hold the accumulated sum of the first n terms.

$$a_n = \frac{(4n)!\,(1103+26390n)}{(n!)^4 396^{4n}}$$

terms	a_n	Σ	1/π	π	Error
0	1.10300E+03	1103.000000000	0.31831	3.14159273001331	7.64E-08
1	2.68320E-05	1103.000026832	0.31831	3.14159265358979	4.44E-16

From the table above we get approx. eight correct digits after the first term and eight more digits that are correct after the second term. This means if you want to as an example calculate π with 1,000 digits you need approx. 1,000/8=125 terms of the Ramanujan series.

To make an effective recurrence of the above formula we need to do some adjustments:

$$\frac{1}{\pi} = \frac{2\sqrt{2}}{9801}\sum_{n=0}^{\infty}\frac{(4n)!\,(1103+26390n)}{(n!)^4 396^{4n}} =>$$

$$\frac{1}{\pi} = \frac{2\sqrt{2}}{9801}\left(1103\sum_{n=0}^{\infty}\frac{(4n)!}{(n!)^4 396^{4n}} + 26390\sum_{n=0}^{\infty}\frac{(4n)!}{(n!)^4 396^{4n}}n\right)$$

Let $a_n = \frac{(4n)!}{(n!)^4 396^{4n}}$ and you get:

$$\frac{1}{\pi} = \frac{2\sqrt{2}}{9801}(1103\sum_{n=0}^{\infty} a_n + 26390\sum_{n=0}^{\infty} a_n\, n)$$

Substitute $b_n = a_n n$ into the above and you get:

$$\pi = \frac{9801}{2\sqrt{2}(1103\sum_{n=0}^{\infty} a_n + 26390\sum_{n=0}^{\infty} b_n)}$$

The only thing left is to calculate as a_n part of the recurrence. We can do that by evaluating:

$$\frac{a_n}{a_{n-1}} = \frac{\frac{(4n)!}{(n!)^4 396^{4n}}}{\frac{(4(n-1))!}{((n-1)!)^4 396^{4(n-1)}}} \quad =>$$

$$\frac{a_n}{a_{n-1}} = \frac{4n(4n-1)(4n-2)(4n-3)}{n^4 396^4} \quad =>$$

$$a_n = \frac{(4n-1)(4n-2)(4n-3)}{6147814464n^3} a_{n-1}$$

We can now write up the complete recurrence:

Let the initial conditions be $a_0 = 1, a_{sum} = 0, b_{sum} = 0$ then for n=1,2,3… you get:

$$a_n = \frac{(4n-1)(4n-2)(4n-3)}{6147814464n^3} a_{n-1}$$

$$a_{sum} += a_n$$

$$b_{sum} += a_n n$$

$$\pi = \frac{9801}{2\sqrt{2}(1103a_{sum} + 26390b_{sum})}$$

Algorithm 2.1 Ramanujan Infinite series

```
// Ramanujan series that add 8 more correct digits per iteration
float_precision pi_ramanujan(uintmax_t digits)
	{
	const unsigned int extra = 2;
	uintmax_t n;
	const size_t precision = digits + 1;
	const float_precision c1103(1103), c26390(26390);	// constants
```

```
const float_precision c9801(9801),c6147814464(6147814464);    // constants
float_precision pi(3, precision);
float_precision an(1,precision+extra), asum(1,precision+extra),
                bsum(0,precision+extra);
float_precision c2sq2(0,precision);
float_precision tmp(0, precision); float_precision np3(0, precision);

c2sq2 = _float_table(_SQRT2, precision + 1);
c2sq2.adjustExponent(1); // multiply by2
for (n=1; ; ++n )
      {
      uintmax_t n4 = 4 * n;
      // Use 64-bit integer arithmetic when possible. Assuming that n<~2G
      // iterations are required
      if (n <= 2'479'700'524)
            np3 += float_precision(n*(3 * n - 3) + 1);
      if (n < 660'562)  // Up to 64bit arithmetic is safe
            tmp = float_precision((n4 - 1)*(n4 - 2)*(n4 - 3));
      else
            {
            tmp = float_precision(n4 - 1);
            tmp *= float_precision(n4 - 2);
            tmp *= float_precision(n4 - 3);
            }
      an *= tmp / (c6147814464 * np3 );
      asum += an;
      if (asum + an == asum)
            break;
      bsum += an * n;
      }

pi = c9801 / (c2sq2*(asum * c1103 + bsum* c26390));
return pi;
}
```

Chudnovsky brothers

The Chudnovsky brother found a variation of the Ramanujan infinite series for π in 1989 using the infinite series:

$$\frac{1}{\pi} = 12\sum_{n=0}^{\infty}\frac{(-1)^n(6n)!\,(13591409+545140134n)}{(3n)!\,(n!)^3 640320^{3n+3/2}}$$

For each term in the series, it produces approx. 14 more correct digits of π, which is six digits more than the Ramanujan series per term. In 1994, the formula was used to calculate approx. 4 billion digits of π. Again in 2010 and 2011, it reach 10 billion digits of π, and finally in 2022 100 trillion digits of π.
In the table below, we use column 'a_n' to denote the n^{th} term of the summations and the column ∑ to hold the accumulated sum of the first n terms.

$$a_k = \frac{(-1)^n (6n)!\,(13591409 + 545140134n)}{(3n)!\,(n!)^3 640320^{4n+3/2}}$$

terms	a_n	Σ	1/π	π	Error
0	1.3591E+07	13591409	0.318309886183797	3.14159265358973	5.91E-14
1	-2.5538E-07	13591409	0.318309886183791	3.14159265358979	0

From the above table, we get approx. 14 correct digits after the first term and 14 more correct digits after the second term (which is outside the limit of the accuracy of the calculation since we only have approx. 15 digits of accuracy). This means if you want to as an example calculate π with 1000 digits you need approx. 1,000/14=72 terms of the Chudnovsky brother algorithm.
As we did for the Ramanujan series, we need to rewrite the above equations as follows:

$$\frac{1}{\pi} = 12 \sum_{n=0}^{\infty} \frac{(-1)^n (6n)!\,(13591409 + 545140134n)}{(3n)!\,(n!)^3 640320^{3n+3/2}} =>$$

$$\frac{1}{\pi} = \frac{12}{640320^{\frac{3}{2}}} (13591409 \sum_{n=0}^{\infty} \frac{(-1)^n (6n)!}{(3n)!\,(n!)^3 640320^{3n}} + 545140134 \sum_{n=0}^{\infty} \frac{(-1)^n (6n)!}{(3n)!\,(n!)^3 640320^{3n}} n)$$

Letting $a_n = \frac{(-1)^n (6n)!}{(3n)!(n!)^3 640320^{3n}}$ you get:

$$\frac{1}{\pi} = \frac{12}{640320^{\frac{3}{2}}} \left(13591409 \sum_{n=0}^{\infty} a_n + 545140134 \sum_{n=0}^{\infty} a_n n \right) =>$$

$$\frac{1}{\pi} = \frac{1359409 \sum_{n=0}^{\infty} a_n + 54514013 \sum_{n=0}^{\infty} a_n n}{426880\sqrt{100005}} =>$$

$$\pi = \frac{426880\sqrt{10005}}{13591409 \sum_{n=0}^{\infty} a_n + 545140134 \sum_{n=0}^{\infty} a_n n}$$

We just need now to find an easier way to calculate a_n.
Letting:

$$\frac{a_n}{a_{n-1}} = \frac{\frac{(-1)^n(6n)!}{(3n)!\,(n!)^3 640320^{3n}}}{\frac{(-1)^{n-1}(6(n-1))!}{(3(n-1))!\,((n-1)!)^3 640320^{3(n-1)}}} \Rightarrow$$

$$\frac{a_n}{a_{n-1}} = -\frac{(6n-5)(6n-4)(6n-3)(6n-2)(6n-1)6n}{3n(3n-1)(3n-2)n^3 640320^3} \Rightarrow$$

$$a_n = -\frac{24(6n-5)(2n-1)(6n-1)}{n^3 640320^3} a_{n-1}$$

We can now write up the complete recurrence:

Let the initial conditions be $a_0 = 1, a_{sum} = 0, b_{sum} = 0$ then for n=1,2,3… you get:

$$a_n = -\frac{24(6n-5)(2n-1)(6n-1)}{640320^3 n^3} a_{n-1}$$

$$a_{sum} += a_n$$

$$b_{sum} += a_n n$$

$$\pi = \frac{426880\sqrt{10005}}{13591409 a_{sum} + 545140134 b_{sum}}$$

Algorithm 2.2 Chudnovsky Infinite series

```
// Chudnovsky series that add 14 more correct digits per iteration
float_precision pi_chudnovsky(uintmax_t digits)
	{
	const unsigned int extra = 2;
	const size_t precision = digits + 1;
	uintmax_t n;
	const float_precision c24(24), c426880(426880), c1005(10005, precision);
	const float_precision c13591409(13591409), c545140134(545140134);
	float_precision pi(3, precision);
	float_precision an(1,precision+extra), asum(1,precision+extra);
	float_precision bsum(0,precision+extra);
	float_precision tmp(0,precision), np3(0,precision);
	float_precision c3_over24(640320,precision);

	c3_over24 *= c3_over24.square();   // 640320^3
	c3_over24 /= c24;                  // 640320^3/24
	for (n = 1; ; ++n)
		{
		// Use 64-bit integer arithmetic when possible. Assuming less than
		// n<~2G iterations are required
		if (n <= 2'479'700'524)
			np3 += float_precision(n*(3 * n - 3) + 1);
		if (n < 635'130)
			tmp = float_precision((6 * n - 5)*(2 * n - 1)*(6 * n - 1));
```

```
            else
                    {
                    tmp = float_precision(6 * n - 5);
                    tmp *= float_precision(2 * n - 1);
                    tmp *= float_precision(6 * n - 1);
                    }
            tmp /= (c3_over24* np3);
            an *= -tmp;
            if (asum + an == asum)
                    break;
            asum += an;
            bsum += an * n;
            }

      pi = c426880 * sqrt(c1005)/(asum * c13591409 + bsum* c545140134);
      return pi;
      }
```

Borwein Brothers

The Borwein brothers presented two infinite series in 1989 and 1993. The 1989 produced 25 correct digits per iteration while the 1993 infinite series produces 50 correct digits per iteration. We will call it the Borwein 25 and Borwein 50 Infinite series.

Borwein 25

Borwein brothers presented an infinite series for calculating π, which was published in 1989. This is similar to the Chudnovsky brother but uses different constants and produces 25 correct digits for π per iteration.

$$\frac{1}{\pi} = 12\sum_{n=0}^{\infty}\frac{(-1)^n(6n)!\,(A+nB)}{(3n)!\,(n!)^3 C^{n+1/2}}$$

Where the constants A, B, and C are:

$$A = 212175710912\sqrt{61} + 1657145277365$$

$$B = 13773980892672\sqrt{61} + 107578229802750$$

$$C = (159999840\sqrt{61} + 1249638720)^3$$

Due to the $\sqrt{61}$ the constant is now a floating-point constant and not an integer constant as for the Ramanujan and Chudnovsky infinite series.
Each additional term will generate approximately 25 correct digits. As expected we only need to calculate the first term n=0 to be correct in our limitation of precision of approx. 15 digits. This means if you want to as an example calculate π with 1,000 digits you need approx. 1,000/25=40 terms of the Borwein brothers algorithm.

Practical implementation of π Algorithms

In the table below, we use column 'a_n' to denote the n^{th} term of the summations and the column ∑ to hold the accumulated sum of the first n terms

$$a_n = \frac{(-1)^n (6n)!\,(A+nB)}{(3n)!\,(n!)^3 C^{n+1/2}}$$

terms	a_n	Σ	1/π	Π	Error
0	0.026526	0.026525824	0.318309886	3.14159265358979	0

As we did for the Ramanujan and Chudnovsky series, we need to rewrite the above equations to create an efficient algorithm as follows:

$$\frac{1}{\pi} = 12\sum_{n=0}^{\infty} \frac{(-1)^n (6n)!\,(A+nB)}{(3n)!\,(n!)^3 C^{n+1/2}} =>$$

$$\frac{1}{\pi} = \frac{12}{\sqrt{C}}\left(\sum_{n=0}^{\infty} \frac{(-1)^n (6n)!}{(3n)!\,(n!)^3 C^n} A + \sum_{n=0}^{\infty} \frac{(-1)^n (6n)!}{(3n)!\,(n!)^3 C^n} Bn\right)$$

Letting $a_n = \frac{(-1)^n (6n)!}{(3n)!(n!)^3 C^n}$ you get:

$$\frac{1}{\pi} = \frac{12}{\sqrt{C}}\left(A\sum_{n=0}^{\infty} a_n + B\sum_{n=0}^{\infty} a_n\, n\right) =>$$

$$\pi = \frac{\sqrt{C}}{12\left(A\sum_{n=0}^{\infty} a_n + B\sum_{n=0}^{\infty} a_n\, n\right)}$$

We just need now to find an easier way to calculate a_n.
Letting:

$$\frac{a_n}{a_{n-1}} = \frac{\dfrac{(-1)^n (6n)!}{(3n)!\,(n!)^3 C^n}}{\dfrac{(-1)^{n-1}(6(n-1))!}{\big(3(n-1)\big)!\,((n-1)!)^3 C^{n-1}}} =>$$

$$\frac{a_n}{a_{n-1}} = -\frac{6n(6n-1)(6n-2)(6n-3)(6n-4)(6n-5)}{3n(3n-1)(3n-2)n^3 C} =>$$

$$a_n = -\frac{24(6n-5)(2n-1)(6n-1)}{n^3 C} a_{n-1}$$

Which is very similar to the result we got with Chudnovsky. Only the constant C differs.

We can now write up the complete Borwein 25 recurrence:

Let the initial conditions be $a_0 = 1, a_{sum} = 0, b_{sum} = 0$ then for n=1,2,3… you get:

$$a_n = -\frac{24(6n-5)(2n-1)(6n-1)}{Cn^3}a_{n-1}$$

$$a_{sum} += a_n$$

$$b_{sum} += a_n n$$

$$\pi = \frac{\sqrt{C}}{12(Aa_{sum} + Bb_{sum})}$$

Where:

$$A = 212175710912\sqrt{61} + 1657145277365$$
$$B = 13773980892672\sqrt{61} + 107578229802750$$
$$C = (159999840\sqrt{61} + 1249638720)^3$$

Algorithm 2.3 Borwein 25

```
// Borwein series that add 25 more correct digits per iteration
float_precision pi_borwein25(unsigned int digits)
	{
	const unsigned int extra = 3;
	const size_t precision = digits + 1;
	uintmax_t n;
	const float_precision c212175710912(212175710912),
c13773980892672(13773980892672);	// constants
	const float_precision c12(12), c159999840(159999840);	// constants
	float_precision A(1657145277365, precision), B(107578229802750, precision),
C(1249638720, digits+extra);
	float_precision c61sq(61, precision+extra);
	float_precision pi(3, precision);
	float_precision an(1, precision+extra), asum(1, precision+extra), bsum(0,
precision+extra);
	float_precision tmp(0, precision), np3(0, precision);

	c61sq = sqrt(c61sq);  // Calculate sqrt(61) at the requested precision
	A += c212175710912 * c61sq;		// Finish up creation of constant A
	B += c13773980892672 * c61sq;// Finish up creation of constant B
	C += c159999840 * c61sq;		// Finish up creation of constant C
	C *= C.square();			// c = c^3;
	// Now iterate until no change in the asum
	for (n = 1; ; ++n)
		{// Use 64-bit integer arithmetic when possible.
		// Assuming less than n<~2G iterations
		if (n <= 2'479'700'524)
			np3 += float_precision(n*(3 * n - 3) + 1);
		if (n < 220'188)
			tmp = float_precision((6 * n - 5)*(48 * n - 24)*(6 * n - 1));
		else
			{// more than wat 64bit can handle
			tmp = float_precision(6 * n - 5);
```

```
                    tmp *= float_precision(48 * n - 24);
                    tmp *= float_precision(6 * n - 1);
                    }
            tmp /= C * np3;
            an *= -tmp;
            if (asum + an == asum)
                    break;
            asum += an;
            bsum += an * n;
            }

    loopcnt_borwein25 = n;
    pi = sqrt(C) / ( (asum * A + bsum* B) * c12 );
    return pi;
    }
```

Borwein 50

In 1993, the Borwein brothers publish another series that find approx. 50 correct digits per iteration.

$$\frac{1}{\pi} = \frac{1}{\sqrt{-C^3}} \sum_{n=0}^{\infty} \frac{(6n)!\,(A+nB)}{(3n)!\,(n!)^3 C^{3n}}$$

Where the constants A, B, and C are:

$$A = 633650283129719995854262 20 + 28337702140800884204682560 0\sqrt{5} + 384\sqrt{5}(10891728551171178200467436212395209160385660 17 + 48709290865788102250773385345416887213512550 40\sqrt{5})^{\frac{1}{2}}$$

$$B = 78499104534966272102897490 00 + 35105866782609320289656064 00\sqrt{5} + 2515968\sqrt{31101}(62602083237890016369933226544440208821 61 + 27996502730604442965772068907188251902 35\sqrt{5})^{\frac{1}{2}}$$

$$C = -214772995063512240 - 96049403338648032\sqrt{5} - 1296\sqrt{5}(10985234579463550323713318473 + 49127462536923627546073959 12\sqrt{5})^{\frac{1}{2}}$$

We notice that since C is negative C^3 is also negative and $-C^3$ is positive so we can do a calculation of $\sqrt{-C^3}$.

Each additional term will generate approximately 50 correct digits. If you want to as an example calculate π with 1000 digits you need approx. 1,000/50=20 terms of the Borwein brothers algorithm.
As we did for the Ramanujan and Chudnovsky series, we need to rewrite the above equations to create an efficient algorithm as follows:

$$\frac{1}{\pi} = \frac{1}{\sqrt{-C^3}} \sum_{n=0}^{\infty} \frac{(6n)!\,(A+nB)}{(3n)!\,(n!)^3 C^{3n}} =>$$

$$\frac{1}{\pi} = \frac{1}{\sqrt{-C^3}} \left(\sum_{n=0}^{\infty} \frac{(6n)!}{(3n)!\,(n!)^3 C^{3n}} A + \sum_{n=0}^{\infty} \frac{(6n)!}{(3n)!\,(n!)^3 C^{3n}} Bn \right)$$

Letting $a_n = \frac{(6n)!}{(3n)!(n!)^3 C^{3n}}$ you get:

$$\frac{1}{\pi} = \frac{1}{\sqrt{-C^3}} \left(A \sum_{n=0}^{\infty} a_n + B \sum_{n=0}^{\infty} a_n\, n \right) =>$$

$$\pi = \frac{\sqrt{-C^3}}{\left(A \sum_{n=0}^{\infty} a_n + B \sum_{n=0}^{\infty} a_n\, n\right)}$$

We just need now to find an easier way to calculate a_n.
Letting:

$$\frac{a_n}{a_{n-1}} = \frac{\frac{(6n)!}{(3n)!\,(n!)^3 C^{3n}}}{\frac{(6(n-1))!}{\big(3(n-1)\big)!\,((n-1)!)^3 C^{3(n-1)}}} =>$$

$$\frac{a_n}{a_{n-1}} = \frac{6n(6n-1)(6n-2)(6n-3)(6n-4)(6n-5)}{3n(3n-1)(3n-2)n^3 C^3} =>$$

$$a_n = \frac{24(6n-5)(2n-1)(6n-1)}{n^3 C^3} a_{n-1}$$

Which is very similar to the result we got with Borwein 1989. Only the constant C differs. (C^3 versus C).

We can now write up the complete Borwein 1993 recurrence:

Let the initial conditions be $a_0 = 1, a_{sum} = 0, b_{sum} = 0$ then for n=1,2,3… you get:

$$a_n = \frac{24(6n-5)(2n-1)(6n-1)}{C^3 n^3} a_{n-1}$$

$$a_{sum} += a_n$$

$$b_{sum} += a_n n$$

$$\pi = \frac{\sqrt{-C^3}}{(Aa_{sum} + Bb_{sum})}$$

Where:

$$A = 633650283129719995854262 20 + 283377021408008842046825600\sqrt{5} + 384\sqrt{5}(10891728551171178200467436212395209160385660 17 + 4870929086578810225077338534541688721351255040\sqrt{5})^{\frac{1}{2}}$$

$$B = 784991045349662721028974900 0 + 351058667826093202896560640 0\sqrt{5} + 2515968\sqrt{31101}(626020832378900163699332265444402088216 1 + 279965027306044429657720689071882519023 5\sqrt{5})^{\frac{1}{2}}$$

$$C = -214772995063512240 - 96049403338648032\sqrt{5} - 1296\sqrt{5}(10985234579463550323713318473 + 491274625369236275460739591 2\sqrt{5})^{\frac{1}{2}}$$

Algorithm 2.4 Borwein 50

```
float_precision pi_borwein50(unsigned int digits)
	{
	const unsigned int extra = 2;
	const size_t precision = digits + 1;
	uintmax_t n;
	float_precision A(0,precision), B(0,precision), C(0,precision+extra),
C3(0,precision+extra);
	float_precision csq5(5, precision);
	float_precision pi(3, precision);
	float_precision an(1, precision+extra), asum(1,precision+extra),
bsum(0,precision + extra);
	float_precision tmp(0,precision), np3(0,precision);

	csq5 = sqrt(csq5);    // Calculate sqrt(5) at the requested precision

	// Build Constant A
	A =  float_precision("633650283129719995854262 20", precision);
	A += float_precision("283377021408008842046825600", precision)*csq5;
	tmp = float_precision("10891728551171178200467436212395209160385660 17",
precision);
	tmp += float_precision("48709290865788102250773385345416887213512550 40",
precision)*csq5;
	tmp = sqrt(tmp);
	A += float_precision(384, precision)*csq5*tmp;

	// Build Constant B
	B = float_precision("784991045349662721028974900 0", precision);
	B +=float_precision("351058667826093202896560640 0", precision)*csq5;
	tmp = float_precision("626020832378900163699332265444402088216 1", precision);
	tmp += float_precision("279965027306044429657720689071882519023 5",
precision)*csq5;
	tmp = sqrt(tmp);
	B += float_precision(2515968, precision)*sqrt(float_precision(3110,
precision))*tmp;

	// Build Constant C
	C = float_precision(-214772995063512240, precision);
	C += float_precision(-96049403338648032, precision)*csq5;
	tmp = float_precision("10985234579463550323713318473", precision);
	tmp += float_precision("491274625369236275460739591 2", precision)*csq5;
```

```
        tmp = sqrt(tmp);
        C += float_precision(-1296, precision)*csq5*tmp;

        // Build Constant C3=C^3
        C3 = C*C.square();           // C3 = C^3;

        // Now iterate until no change in the asum
        for (n = 1; ; ++n)
                {// Use 64-bit integer arithmetic when possible.
                // Assuming less than n<~2G iterations
                if (n <= 2'479'700'524)
                        np3 += float_precision(n*(3*n-3)+1);
                if (n < 220'188)
                        tmp = float_precision((6*n-5)*(48*n-24)*(6*n-1));
                else
                        {// more than wat 64bit can handle
                        tmp = float_precision(6*n-5);
                        tmp *= float_precision(48*n-24);
                        tmp *= float_precision(6*n-1);
                        }
                tmp /= C3 * np3;
                an *= tmp;
                if (asum + an == asum)
                        break;
                asum += an;
                bsum += an * n;
                }

        pi = sqrt(-C3) / (asum * A + bsum* B);
        return pi;
        }
```

The Binary splitting method for Ramanujan and Chudnovsky series

Another method that is now considered the best is to use the Binary splitting method for the Ramanujan or Chudnovsky series.

Binary splitting of the Ramanujan infinite series

Instead of adding each series term we instead try to find two integers P & Q that equates to the first k terms of the series

$$\frac{1}{\pi} = \frac{2\sqrt{2}}{9801}\sum_{n=0}^{\infty}\frac{(4n)!\,(1103+26390n)}{(n!)^4 396^{4n}} =>$$

$$\frac{1}{\pi} = \frac{2\sqrt{2}}{9801}\frac{P}{Q}$$

Given the first k terms of the series, you get (see [15]):

$$\frac{1}{\pi} = \frac{2\sqrt{2}}{9801}\frac{P(0,k) + 1103Q(0,k)}{Q(0,k)} =>$$

$$\pi = \frac{9801Q(0,k)}{P(0,k) + 1103Q(0,k)}\frac{1}{2\sqrt{2}} + O(96059301^{-k})$$

Where k, is found to satisfy the precision of the number. E.g. for precision P we have equality:

$$10^{-p} < 96059301^{-k} => k > \frac{P \cdot log(10)}{\log(96059301)}$$

We take k as the ceiling of $\left\lceil \frac{P \cdot log(10)}{\log(96059301} \right\rceil$

To find the P(0,k) and Q(0,k) you can use a recursive formula for P(a,b) & Q(a,b) and a<b:

$m = \frac{a+b}{2}$ *integer division*

P(a,b)=P(a,m)Q(m,b)+P(m,b)R(a,m)
Q(a,b)=Q(a,m)Q(m,b)
R(a,b)=R(a,m)R(m,b)

Where:
P(b+1,b)=(1103+26390b)(2b-1)(4b-3)(4b-1)
$Q(b-1,b)=3073907232b^3$
R(b-1,b)=(2b-1)(4b-3)(4b-1)

Algorithm 2.5 Binary splitting for Ramanujan π

There are two functions. binarysplittingRamanujanPI() that performs the recursion top-down and the driver functions computeRamanujandigit() that computes the requested number of digits for π. The functions are surprisingly simple.

```
static void binarysplittingRamanujanPI(const uintmax_t a, const uintmax_t b,
int_precision& p, int_precision& q, int_precision& r)
	{
	int_precision pp, qq, rr;
	uintmax_t mid;

	if (b - a == 1)
		{
		if(b<=832'256)// No overflow using 64bit arithmetic
			r = int_precision((2 * b - 1)*(4 * b - 3)*(4 * b - 1));
		else
			{// No overflow if b<=1073'741'825 ~ 8.5B digits
			r = int_precision((4 * b - 3)*(4 * b - 1));
			r *= int_precision(2 * b - 1);
			}
		p = int_precision(1'103ull + 26'390ull * b); p *= r;
		q = int_precision(b); q *= q*q;
```

```
                q *= int_precision(3'073'907'232ull);
                return;
                }
        mid = (a + b) / 2;
        binarysplittingRamanujanPI(a, mid, p, q, r);     // interval [a..mid]
        binarysplittingRamanujanPI(mid, b, pp, qq, rr); // interval [mid..b]
        // Reconstruct interval [a..b] and return p, q & r
        p = p*qq + pp*r;
        q *= qq;
        r *= rr;
        }

// Binary splitting with recursion
float_precision computeRamanujandigit(uintmax_t digits)
        {
        uintmax_t k;
        int_precision p, q, r;
        float_precision pi(0,digits);

        k = (uintmax_t)ceil( digits*log(10) / log(96059301ull) );
        piSplitting = 0;
        binarysplittingRamanujanPI(0, k, p, q, r);
        pi = (float_precision(9801ull) * float_precision(q,digits)) /
(float_precision(p,digits) + float_precision(1103ull)*float_precision(q,digits));
        pi*=1 / sqrt(float_precision(8, digits));
        return pi;
        }
```

Binary splitting of the Chudnovsky infinite series

Instead of adding each series of terms we instead try to find two integers, P & Q that equate to the first k terms of the series.

$$\frac{1}{\pi} = 12\sum_{n=0}^{\infty}\frac{(-1)^n(6n)!\,(13591409+545140134n)}{(3n)!\,(n!)^3 640320^{3n+3/2}} =>$$

$$\frac{1}{\pi} = \frac{\sqrt{10005}}{4270934400}\sum_{n=0}^{\infty}\frac{(-1)^n(6n)!\,(13591409+545140134n)}{(3n)!\,(n!)^3 640320^{3n}} =>$$

$$\frac{1}{\pi} = \frac{\sqrt{10005}}{4270934400}\frac{P}{Q}$$

Given the first k terms of the series, you get (see [15]):

$$\frac{1}{\pi} = \frac{\sqrt{10005}}{4270934400}\frac{P(0,k)+13591409Q(0,k)}{Q(0,k)} =>$$

$$\pi = \frac{4270934400\cdot Q(0,k)}{P(0,k)+13591409Q(0,k)}\frac{1}{\sqrt{10005}} + O(151931373056000^{-k})$$

Where k, is found to satisfy the precision of the number. E.g. for precision P we have equality:

$$10^{-p} < 151931373056000^{-k} => k > \frac{P \cdot log(10)}{\log(151931373056000)}$$

We take k as the ceiling of $\left\lceil \frac{P \cdot log(10)}{\log(151931373056000} \right\rceil$

To find the P(0,k) and Q(0,k) you can use a recursive formula for P(a,b) & Q(a,b) and a<b:

$m = \frac{a+b}{2}$ *integer division*

P(a,b)=P(a,m)Q(m,b)+P(m,b)R(a,m)
Q(a,b)=Q(a,m)Q(m,b)
R(a,b)=R(a,m)R(m,b)

Where:
P(b+1,b)=(13591409+545140134b)(2b-1)(6b-5)(6b-1)$(-1)^b$
Q(b-1,b)=10939058860032000b^3
R(b-1,b)= (2b-1)(6b-5)(6b-1)

Algorithm 2.6 Binary splitting for Chudnovsky π

There are two functions. binarysplittingChudnovskyPI() that performs the recursion top-down and the driver functions computeChudnovskydigit() that computes the requested number of digits for π. The functions are surprisingly simple:

```
static void binarysplittingChudnovskyPI(const uintmax_t a, const uintmax_t b,
int_precision& p, int_precision& q, int_precision& r)
	{
	int_precision pp, qq, rr;
	uintmax_t mid;

	if (b - a == 1)
		{// No overflow using 64bit arithmetic
		if (b<635'130)// No overflow using 64bit arithmetic
			r = int_precision((2 * b - 1)*(6 * b - 5)*(6 * b - 1));
		else
			{// No overflow if b<=715'827'883 ~ 10B digits
			r = int_precision((6 * b - 5)*(6 * b - 1));
			r *= int_precision(2 * b - 1);
			}
		p = int_precision(13'591'409ull + 545'140'134ull * b); p *= r;
		if (b & 0x1)
			p.change_sign();
		q = int_precision(b); q *= q*q;
		q *= int_precision(10'939'058'860'032'000ull);
		return;
		}
	mid = (a + b) / 2;
	binarysplittingChudnovskyPI(a, mid, p, q, r);   // interval [a..mid]
	binarysplittingChudnovskyPI(mid, b, pp, qq, rr);// interval [mid..b]
	// Reconstruct interval [a..b] and return p, q & r
	p = p*qq + pp*r;
```

```
        q *= qq;
        r *= rr;
        }

// Binary splitting with recursion
float_precision computeChudnovskydigit(uintmax_t digits)
        {
        uintmax_t k;
        int_precision p, q, r;
        float_precision pi(0,digits);

        k = (uintmax_t)ceil(digits*log(10) / log(151931373056000ull));
        piSplitting = 0;
        binarysplittingChudnovskyPI(0, k, p, q, r);
        pi = (float_precision(4270934400ull) * float_precision(q, digits)) /
(float_precision(p, digits) + float_precision(13591409ull)*float_precision(q,
digits));
        pi *= 1 / sqrt(float_precision(10005, digits));
        return pi;
        }
```

Binary splitting of the Borwein25 infinite series

Borwein25 infinite series is given by:

$$\frac{1}{\pi} = 12\sum_{n=0}^{\infty}\frac{(-1)^n(6n)!\,(A+nB)}{(3n)!\,(n!)^3 C^{n+1/2}} =>$$

$$\frac{1}{\pi} = \frac{12}{\sqrt{C}}\frac{P}{Q}$$

Given the first k terms you get (see Appendix)

$$\pi = \frac{\sqrt{C}\cdot Q(0,k)}{12\cdot(P(0,k)+A\cdot Q(0,k))}$$

To find the P(0,k) and Q(0,k) you can use a recursive formula for P(a,b) & Q(a,b) and a<b: Please note that since the constant A, B, C & D is no longer integers (as it was for the Ramanujan and Chudnovsky series) P(a,b), Q(a,b) is arbitrary precision floating point numbers, however, R(a,b) can still be kept as arbitrary precision integers)

$m = \frac{a+b}{2}$ *integer division*

P(a,b)=P(a,m)Q(m,b)+P(m,b)R(a,m)
Q(a,b)=Q(a,m)Q(m,b)
R(a,b)=R(a,m)R(m,b)

Where:
P(b+1,b)= (A+Bb)(2b-1)(6b-5)(6b-1)$(-1)^b$

Q(b-1,b)=Db^3
R(b-1,b)= (2b-1)(6b-5)(6b-1)

Where A, B, C is the constant from the description of Borwein25 infinite series and $D = \frac{C}{24}$.

Algorithm 2.7 Binary splitting for Borwein25 π

```
static void binarysplittingBorwein25PI(const uintmax_t a, const uintmax_t b,
float_precision& p, float_precision& q, int_precision& r, float_precision& A,
float_precision& B, float_precision& C24)
	{
	float_precision pp(0,p.precision()), qq(0,q.precision());
	int_precision rr;
	uintmax_t mid;

	if (b - a == 1)
		{// No overflow using 64bit arithmetic
		if (b<635'130)// No overflow using 64bit arithmetic
			r = int_precision((2 * b - 1)*(6 * b - 5)*(6 * b - 1));
		else
			{// No overflow if b<=715'827'883 ~ 17B digits
			r = int_precision((6 * b - 5)*(6 * b - 1));
			r *= int_precision(2 * b - 1);
			}
		p = A + B * float_precision(b); p *= r;
		if (b & 0x1)
			p.change_sign();
		q= float_precision(b); q *= q * q;
		q *= C24;
		return;
		}
	mid = (a + b) / 2;
	binarysplittingBorwein25PI(a, mid, p, q, r, A, B, C24);   // interval [a..mid]
	binarysplittingBorwein25PI(mid, b, pp, qq, rr, A, B, C24);// interval [mid..b]

	// Reconstruct interval [a..b] and return p, q & r
	p = p*qq + pp*float_precision(r,pp.precision());
	q *= qq;
	r *= rr;
	}

// Binary splitting with recursion
static float_precision computeBorwein25digit(uintmax_t digits)
	{
	const uintmax_t precision = std::max((uintmax_t)20,digits);
	uintmax_t k;
	int_precision r;
	const float_precision c212175710912(212175710912),
c13773980892672(13773980892672);     // constants
	const float_precision c12(12), c159999840(159999840);
                          // constants
	float_precision p(0, precision), q(0, precision), pi(0, precision);
	float_precision A(1657145277365, precision), B(107578229802750, precision),
C(1249638720, precision);
	float_precision Cover24(24, precision), c61sq(61,precision);

	k = (uintmax_t)ceil(precision / 24) + 1;
```

```
        piSplitting = 0;
        c61sq = sqrt(c61sq);  // Calculate sqrt(61) at the requested precision
        A += c212175710912 * c61sq;          // Finish up creation of constant A
        B += c13773980892672 * c61sq;// Finish up creation of constant B
        C += c159999840 * c61sq;             // Finish up creation of constant C
        C *= C.square();
        Cover24 = C / Cover24;
        binarysplittingBorwein25PI(0, k, p, q, r, A, B, Cover24);
        pi = sqrt(C) * q / (c12 * (p + A * q));
        pi.precision(digits);
        return pi;
        }
```

Binary splitting of the Borwein50 infinite series

Borwein50 Infinite series is given by:

$$\frac{1}{\pi} = \frac{1}{\sqrt{-C^3}} \sum_{n=0}^{\infty} \frac{(6n)!\,(A+nB)}{(3n)!\,(n!)^3 C^{3n}}$$

Given the first k terms, you get (see Appendix):

$$\pi = \frac{\sqrt{-C^3} \cdot Q(0,k)}{(P(0,k) + A \cdot Q(0,k))}$$

To find the P(0,k) and Q(0,k) you can use a recursive formula for P(a,b) & Q(a,b) and a<b: Please note that since the constant A, B, C & D is no longer integers (as it was for the Ramanujan and Chudnovsky series) P(a,b), Q(a,b) is arbitrary precision floating point numbers, however, R(a,b) can still be kept as arbitrary precision integers)

$m = \frac{a+b}{2}$ *integer division*

P(a,b)=P(a,m)Q(m,b)+P(m,b)R(a,m)
Q(a,b)=Q(a,m)Q(m,b)
R(a,b)=R(a,m)R(m,b)

Where:
P(b+1,b)= (A+Bb)(2b-1)(6b-5)(6b-1)$(-1)^b$
Q(b-1,b)=Db^3
R(b-1,b)= (2b-1)(6b-5)(6b-1)

Where A, B, C is the constant from the description of Borwein50 infinite series and $D = \frac{C^3}{24}$.

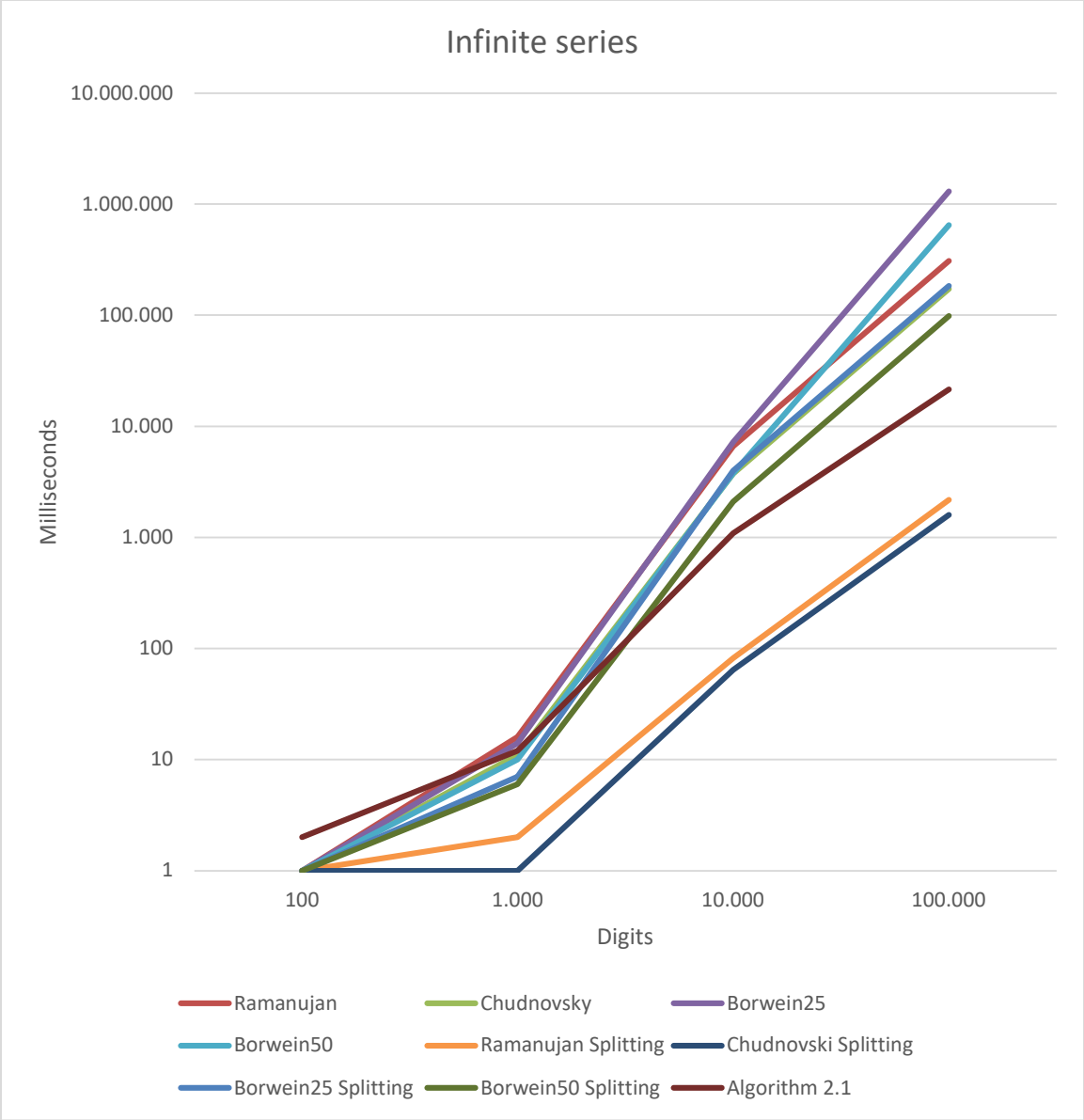
Algorithm 2.8 Binary splitting for Borwein50 π

```
static void binarysplittingBorwein50PI(const uintmax_t a, const uintmax_t b,
float_precision& p, float_precision& q, int_precision& r, float_precision& A,
float_precision& B, float_precision& C24)
	{
	float_precision pp(0, p.precision()), qq(0, q.precision());
	int_precision rr;
	uintmax_t mid;

	if (b - a == 1)
		{// No overflow using 64bit arithmetic
		if (b<635'130)// No overflow using 64bit arithmetic
			r = int_precision((2 * b - 1)*(6 * b - 5)*(6 * b - 1));
		else
			{// No overflow if b<=715'827'883 ~ 34B digits
			r = int_precision((6 * b - 5)*(6 * b - 1));
			r *= int_precision(2 * b - 1);
			}
		p = A + B * float_precision(b); p *= r;
		q = float_precision(b); q *= q * q;
		q *= C24;
		return;
		}
	mid = (a + b) / 2;
	binarysplittingBorwein50PI(a, mid, p, q, r, A, B, C24);   // interval [a..mid]
	binarysplittingBorwein50PI(mid, b, pp, qq, rr, A, B, C24);// interval [mid..b]

	// Reconstruct interval [a..b] and return p, q & r
	p = p*qq + pp*float_precision(r, pp.precision());
	q *= qq;
	r *= rr;
	}

// Binary splitting with recursion
static float_precision computeBorwein50digit(uintmax_t digits)
	{
	const uintmax_t precision = std::max((uintmax_t)20, digits);
	uintmax_t k;
	int_precision r;
	float_precision p(0, precision), q(0, precision), pi(0, precision);
	float_precision A(0, precision), B(0, precision), C(0, precision);
	float_precision Cover24(24, precision), csq5(5, precision), tmp(0,precision);

	k = (uintmax_t)ceil(precision / 49) + 1;
	piSplitting = 0;

	csq5 = sqrt(csq5);           // Calculate sqrt(5) at the requested precision
	// Build Constant A
	A = float_precision("6336502831297199585426220", precision);
	A += float_precision("2833770214080084206825600", precision)*csq5;
	tmp = float_precision("10891728551171178200467436212395209160385656017",
precision);
	tmp += float_precision("4870929086578810225077338534541688721351255040",
precision)*csq5;
	tmp = sqrt(tmp);
	A += float_precision(384, precision)*csq5*tmp;
	// Build Constant B
	B = float_precision("784991045349662721028974900O", precision);
	B += float_precision("351058667826093202896560640O", precision)*csq5;
	tmp = float_precision("626020832378900163699332265444402088216l", precision);
```

```
        tmp += float_precision("279965027306044429657720689071882519023 5",
precision)*csq5;
        tmp = sqrt(tmp);
        B += float_precision(2515968, precision)*sqrt(float_precision(3110,
precision))*tmp;
        // Build Constant C
        C = float_precision(-214772995063512240, precision);
        C += float_precision(-96049403338648032, precision)*csq5;
        tmp = float_precision("10985234579463550323713318473", precision);
        tmp += float_precision("491274625369236275460739591 2", precision)*csq5;
        tmp = sqrt(tmp);
        C += float_precision(-1296, precision)*csq5*tmp;
        // Build Constant C3=C^3
        C = C*C.square();            // C = C^3;
        Cover24 = C / Cover24;
        binarysplittingBorwein50PI(0, k, p, q, r, A, B, Cover24);
        pi = sqrt(-C) * q / (p + A * q);
        pi.precision(digits);
        return pi;
        }
```

Speed Comparison of the Infinite series

Speed comparison of the Ramanujan, Chudnovsky, and Borwein to the standard method



As can be seen nearly identical performance with an edge on the Chudnovsky algorithm that produces 14 more correct digits compared to Ramanujan with 8 digits per series and Borwein brother with 25 & 50 digits per series. It is a surprise that the Borwein series despite the nearly 2-4 as many digits as the Chudnovsky per series still are significantly slower than Chudnovsky, despite being similar. However, more surprisingly s the edge the Binary splitting method has for the Ramanujan and the Chudnovsky series. It is approx. 100-150 times faster that their corresponding Infinite series implementation. Even when using binary splitting for Borwein25 and Borwein50 it does not approach the performance of the Chudnovky and Ramanujan Binary splitting method although it is a huge performance gain over the traditional Series for Borwein25 and Borwein50.

Recommendation for the Infinite series

I recommend always using the binary spitting algorithm for Chudnovsky, which has the fastest performance of them all. It is no surprise that it is the Chudnovsky binary splitting method that is used in the record-breaking calculation of π with 100 trillion digits (2022).

Higher order algorithm for π

In this section, we will describe some higher orders methods that were developed from the seventies up to the end of the nineties. Particular worth mentioning is the Borwein brothers that discover many new algorithms of quadratic to must higher order of convergence.

Brent-Salamin method for π

This is another famous algorithm for quickly calculating π. First published in 1976 by Brent. It uses the arithmetic-geometric mean to calculate π. The convergence rate is quadratic mean for each iteration it has twice as many correct digits compared to the previous iteration.

It uses the following recursion, starting with the initial conditions:

$$a_0 = 1, b_0 = \frac{1}{\sqrt{2}}, c_0 = 0.5$$

Then repeat for n=0,1,2… until sufficient accuracy has been obtained.

$$a_{n+1} = \frac{1}{2}(a_n + b_n)$$

$$b_{n+1} = \sqrt{a_n b_n}$$

$$c_{n+1} = c_n - 2^{n+1}(a_{n+1} - b_n)^2$$

$$\pi_{n+1} = 2\frac{a_{n+1}^2}{c_{n+1}}$$

In [10] Borwein has $c_{n+1} = c_n - 2^{n+1}(a_{n+1}^2 - b_{n+1}^2)$ which is the same as the c_{n+1} above as shown here:

$$c_n - 2^{n+1}(a_{n+1} - b_n)^2 =$$

$$c_n - 2^{n+1}(a_{n+1}^2 + b_n^2 - 2a_{n+1}b_n) =$$

$$c_n - 2^{n+1}(a_{n+1}^2 + b_n^2 - 2(\frac{a_n + b_n}{2})b_n) =$$

$$c_n - 2^{n+1}(a_{n+1}^2 + b_n^2 - b_n^2 - a_n b_n) =$$

$$c_n - 2^{n+1}(a_{n+1}^2 - a_n b_n) =$$

$$c_n - 2^{n+1}(a_{n+1}^2 - b_{n+1}^2)$$

An iteration looks like this and after four iterations; we have reached π to 15 digits of accuracy

Brent-Salami

Iteration	a	b	c	π	Error
0	1	0.707107	0.5		8.58E-01
1	0.853553	0.840896	0.457107	3.18767264271211	4.61E-02
2	0.847225	0.847201	0.456947	3.14168029329765	8.76E-05
3	0.847213	0.847213	0.456947	3.14159265389545	3.06E-10
4	0.847213	0.847213	0.456947	3.14159265358979	8.88E-16

Algorithm 3.1 Brent-Salamin

```
// Begin Brent-Salamin
float_precision pi_brent_salamin(uintmax_t digits)
	{
	const size_t min_precision = digits + 5 + (size_t)(log10(digits) + 0.5);
	const eptype limit = -(eptype)ceil((digits+2)*log2(10));
	float_precision a(1, min_precision), b(2, min_precision), sum(0.5,
min_precision);
	float_precision ak(0, min_precision), bk(0, min_precision), ck(1,
min_precision);
	float_precision ab(0, min_precision), asq(0, min_precision);
	float_precision pow2(1,digits), pi(3,digits);
	const float_precision c1(1), c2(2);

	b = c1 / sqrt(b);
	for ( ; !ck.iszero() && ck.exponent()>limit; )
		{
		ak = a + b;
		ak.adjustExponent(-1); // ak*=0.5
		ab = a * b;
		bk = sqrt(ab);
		asq = ak.square();
		ck = asq - ab;
		pow2.adjustExponent(+1);  // pow2*=2
		sum -= pow2*ck;
		a = ak; b = bk;
		}

	pi = c2 * asq / sum;
	return pi;
	}
```

Practical implementation of π Algorithms

Gauss-Legendre method for π

Another algorithm and equally as good as the Brent-Salamin is the Gauss-Legendre which is a deviation of the Bent-Salamin algorithm.

The Gauss-Legendre starts with the initial settings:

$$a_0 = 1, b_0 = \frac{1}{\sqrt{2}}, t_0 = 0.25$$

Then repeat for n=0,1,2,3,…

$$a_{n+1} = \frac{1}{2}(a_n + b_n)$$

$$b_{n+1} = \sqrt{a_n b_n}$$

$$t_{n+1} = t_n - 2^n(a_n - a_{n+1})^2$$

$$\pi_{n+1} = \frac{(a_{n+1} + b_{n+1})^2}{4t_{n+1}}$$

Gauss-Legendre

Iteration	a	b	t	π	Error
0	1	0.707107	0.25	2.91421356237309	2.27E-01
1	0.853553	0.840896	0.228553	3.14057925052217	1.01E-03
2	0.847225	0.847201	0.228473	3.14159264621354	7.38E-09
3	0.847213	0.847213	0.228473	3.14159265358979	8.88E-16

The result is nearly identical to the Brent-Salamin algorithm.

Algorithm 3.2 Gauss-Legendre

```
// Gauss-Legendre
float_precision pi_gauss_legendre(uintmax_t digits)
	{
	const size_t min_precision = digits+5+(size_t)(log10(digits)+0.5);
	const int limit = -(int)(digits+2);
	const float_precision c0(0), c1(1), c2(2), c4(4);
	float_precision a(1,min_precision), b(2,min_precision), pow2(0.5,digits);
	float_precision ak(0,min_precision), bk(0,min_precision),
tk(0.25,min_precision);
	float_precision pi(3,digits), tmp(0,min_precision), sq(1,min_precision);

	b = c1 / sqrt(b);
	for ( ; sq != c0 && sq.exponent()>limit; )
		{
		ak = a + b;
		ak.adjustExponent(-1); // ak*=0.5
		bk = sqrt(a * b);
		pow2.adjustExponent(+1); // pow2*=2
```

```
            sq = (a - ak);
            tk -= pow2*sq.square();
            a = ak; b = bk;
            }

        tmp = ak + bk;
        pi = (tmp * tmp) / (c4 * tk);
        return pi;
        }
```

Borwein Brothers Algorithms for PI

The Borwein brothers did a lot of research on the subject and have written an excellent book [4] "PI and the AGM" about it where they revealed several interesting algorithms for calculating π. Furthermore, they have over the years published several articles. Reference [8], [9], [10], [11] & [12] are great papers on the subject, that also include a history walkthrough on the progress of calculating π from the ancient time up to the modern time.

Borwein Quadratic Algorithm 2.1 from 1987

One of the early ones is their quadratic algorithm from 1987 (Algorithm 2.1 from [4]) derived from the Gauss Arithmetic Geometric Mean iteration (AGM).

Let:

$$x_0 = \sqrt{2}, y_0 = \sqrt[4]{2}, \pi_0 = 2 + \sqrt{2}$$

Now iterate through n=1,2,3…

$$x_n = \frac{1}{2}\left(\sqrt{x_{n-1}} + \frac{1}{\sqrt{x_{n-1}}}\right)$$

$$y_n = \frac{y_{n-1}\sqrt{x_n} + \frac{1}{\sqrt{x_n}}}{y_{n-1} + 1}$$

$$\pi_n = \pi_{n-1}\frac{x_n}{y_{n-1}}$$

The algorithm is efficient with a quadratic convergence rate. Borwein state that the first nine iterations give correct 1, 3, 8, 19, 41, 83, 170, 345, and 694 digits of π. Which corresponds exactly with the below result of the first four iterations.

Borwein Quadratic 1984

Iteration	x	π	y	Error
0	1.414214	3.41421356237309	1.189207	0.272621

1	1.015052	3.14260675394162	1.000673	0.001014
2	1.000028	3.14159266096604	1	7.38E-09
3	1	3.14159265358979	1	0

Algorithm 3.3 Borwein Quadratic 2.1

```
// Borwein Quadratic algorithm 2.1
float_precision pi_borwein_algorithm2_1(uintmax_t digits)
	{
	const size_t precision = digits + 2 + (size_t)(log10(digits) + 0.5);
	const float_precision c1(1), c2(2);
	float_precision x(2,precision), y(0,precision), z(0,precision);
	float_precision xsq(0,precision), xsq_inv(0,precision);
	float_precision r(0,precision), rold(1,precision);
	float_precision pi(0,precision);
	size_t loopcnt = 1;

	// Initial setup.
	//	x0 = sqrt(2), pi = 2 + sqrt(2), y = 2^(1/4)
	x = sqrt(x);
	xsq = sqrt(x);
	pi = x + c2;
	xsq_inv = xsq.inverse();
	y = xsq;

	// Iterate.
	//	x = 0.5(sqrt(x)+1/sqrt(x)),
	//	pi=pi((x+1)/(y+1)),
	//	y=(y*sqrt(x)+1/sqrt(x))/(y+1)
	for ( ;;++loopcnt )
		{
		x = (xsq + xsq_inv);
		x.adjustExponent(-1);		// x*= 0.5
		r = (x + c1) / (y + c1);
		pi *= r;
		xsq = sqrt(x);
		xsq_inv = xsq.inverse();
		y = (y * xsq+xsq_inv) / (y + c1);
		// Stop when r==1 or when r doesn't change within an iteration
		if (r == c1 || rold == r)
			break;
		rold = r;
		}

	pi.precision(digits);		// Round to desired precision
	return pi;
	}
```

Borwein Quadratic algorithm 1985

Another algorithm publishes shortly after the 1984 algorithm is the one below that still has quadratic convergence so it is similar in performance to the 1984 algorithm.

Let the initial conditions be: $a_0 = 6 - 4\sqrt{2}, y_0 = \sqrt{2} - 1$

And then for n=1,2,3…

$$f(y_{n-1}) = \sqrt[4]{1 - y_{n-1}^4}$$

$$y_n = \frac{1 - f(y_{n-1})}{1 + f(y_{n-1})}$$

$$a_n = a_{n-1}(1 + y_n)^4 - 2^{2n+1} y_n (1 + y_n + y_n^2)$$

$$\pi_n = \frac{1}{a_n}$$

Iteration	Y_n	$f(y_n)$	a_n	π_n	Error
0	4.14E-01	0.992558024001326	0.343145750507619	2.91421356237310	0.227379
1	3.73E-03	0.999999999951354	0.318309886931161	3.14159264621355	7.38E-09
2	2.43E-11	1.000000000000000	0.318309886183791	3.14159265358979	1.33E-15

This particular algorithm has been used as one of the algorithms for the quest of who could calculate π with the highest amount of digits. In 1986 it was used to compute 29.4 million digits of π, and reach over 6 billion digits in 1995 [8] and in 2009 Takahashi calculated more than 2.5 trillion digits using this algorithm [9]

Algorithm 3.4 Borwein Quadratic 1984

```
// Borwein quadratic algorithm from 1985
float_precision pi_borwein_1985(unsigned int digits)
	{
	const size_t precision = digits + 2 + (size_t)(log10(digits) + 0.5);
	const float_precision c1(1), c6(6);
	float_precision a(2, precision), y(2, precision);
	float_precision pow2(2, precision), fy(0, precision);
	float_precision an(0, precision), yn(1, precision),  yn1(0,precision);
	float_precision pi(0, precision), tmp(0,precision);
	size_t loopcnt = 1;

	// Initial Setup
	a = sqrt(a);                        // sqrt(2)
	y = a - c1;                                 // sqrt(2)-1
	a.adjustExponent(+2);               // 4*sqrt(2)
	a = c6 - a;                                 // 6-4*sqrt(2)
	for ( ; ;++loopcnt )
		{
		fy = y.square();            // y^2
		fy = fy.square();           // y^4
		fy = c1 - fy;               // 1 - y^4
		fy = nroot(fy, 4);          //(1-y^4)^(1/4)
		yn = (c1-fy)/(c1+fy);       // (1-f(y))/(1+f(y))
		// stop if yn ==0 => an==a and no further improvement
		if (yn.iszero())
			break;
```

```
            pow2.adjustExponent(+2);// pow2*=4;  2^(2n+1)
            yn1 = c1 + yn;                      // 1+y
            tmp = yn1.square();          // (1+y)^2
            tmp = tmp.square();          // (1+y)^4
            an = a*tmp - pow2*yn*(yn1 + yn*yn);  //
a=a*(1+y)^4+2^(2n+1)(1+y+y^2)
            a = an;
            y = yn;
            }

      pi = an.inverse();                 // pi = 1/a
      pi.precision(digits);              // Round to precision
      return pi;
      }
```

Borwein Quadratic 1984

Another Quadratic algorithm from 1984. The iteration goes as follows:

Let the initial conditions be: $x_0 = \sqrt{2},\ y_0 = 0,\ \pi_0 = 2 + \sqrt{2}$

Then iterate through n=1,2,3…

$$x_n = \frac{1}{2}\left(\sqrt{x_{n-1}} + \frac{1}{\sqrt{x_{n-1}}}\right)$$

$$y_n = \frac{(1 + y_{n-1})\sqrt{x_{n-1}}}{x_{n-1} + y_{n-1}}$$

$$\pi_n = \pi_{n-1} y_n \frac{x_n + 1}{y_n + 1}$$

π

Iteration	x	y	π	Error
0	1.414214	0	3.41421356237309	0.272621
1	1.015052	0.840896	3.14260675394162	0.001014
2	1.000028	0.999327	3.14159266096604	7.38E-09
3	1	1	3.14159265358979	0

Compare it to Borwein's 1987 algorithm it is identical in the iterations steps.

Algorithm 3.5 Borwein Quadratic 1984

```
// Borwein Quadratic 1984
float_precision pi_borwein_1984(unsigned int digits)
      {
      const size_t precision = digits + 2 + (size_t)(log10(digits) + 0.5);
      const float_precision c1(1), c2(2);
```

```
        float_precision x(2, precision), y(0, precision);
        float_precision xsq(0, precision), xsq_inv(0, precision);
        float_precision r(0, precision), rold(1, precision);
        float_precision pi(0, precision);
        size_t loopcnt = 1;

        // Initial Setup
        // x0 = sqrt(2), pi = 2 + sqrt(2), y = 0
        x = sqrt(x);
        xsq = sqrt(x);
        pi = x + c2;
        xsq_inv = xsq.inverse();
        // Iterate.
        //     y=((1+y)*sqrt(x))/(x+y+1),
        //     x = 0.5(sqrt(x)+1/sqrt(x)),
        //     pi=pi(y(x+1)/(y+1))
        for (;;++loopcnt )
                {
                y = (c1 + y)*xsq/(x+y);
                x = (xsq + xsq_inv);
                x.adjustExponent(-1);  // x*=c05;
                r = (x + c1) / (y + c1);
                r *= y;
                pi *= r;
                xsq = sqrt(x);
                xsq_inv = xsq.inverse();
                // Stop when r gets 1 or r doesn't change within an iteration
                if (r == c1 || rold == r)
                        break;
                rold = r;
                }

        pi.precision(digits);       // Round to desired precision
        return pi;
        }
```

Borwein Cubic 1991

With continuous research into convergence series, the Borwein brothers discover more series with high-order convergence rates that the four mentioned above. The Cubic version was from 1991 and goes as follows:

Let the initial conditions be: $a_0 = \frac{1}{3},\ s_0 = \frac{\sqrt{3}-1}{2}$

And then for n=1,2,3…

$$r_n = \frac{3}{1 + 2\sqrt[3]{1 - s_{n-1}^3}}$$

$$s_n = \frac{r_n - 1}{2}$$

$$a_n = r_n^2 a_{n-1} - 3^{n-1}(r_n^2 - 1)$$

$$\pi_n = \frac{1}{a_n}$$

Then π_n converges cubically meaning for each iteration the number of correct digits triples.

π

Iteration	r	s	a	π	Error
0		0.366025	0.333333	3.0000000000000	0.141593
1	1.011205	0.005602	0.31831	3.1415905852059	2.07E-06
2	1	1.95E-08	0.31831	3.1415926535898	1.11E-14

Algorithm 3.6 Borwein Cubic 1991

```
// Borwein cubic algorithm from 1991
float_precision pi_borwein_cubic_1991(unsigned int digits)
	{
	const size_t precision = digits + 2 + (size_t)(log10(digits) + 0.5);
	bool first_time = true;
	const float_precision c1(1), c3(3);
	float_precision a(3, precision), r(0,precision), s(3, precision);
	float_precision pow3(1, precision), rsq(0, precision);
	float_precision pi(0, precision);
	size_t loopcnt = 1;

	a = a.inverse();                    // 1/3
	s =(sqrt(s) - c1);                  // (sqrt(3)-1)
	s.adjustExponent(-1);               // s*= 0.5
	for ( ; ;++loopcnt )
		{
		r = nroot(c1-s.square()*s, 3);     // (1-s^3)^(1/3)
		// Break if r==1 => no further improvement
		if (r == c1)
			break;                  //Most common break point
		r.adjustExponent(+1);       // 2(1-s^3)^(1/3)
		r += c1;                    // 1+2(1-s^3)^(1/3)
		r = c3 / r;                 // 3/(1+2(1-s^3)^(1/3))
		// Break if r==1 => no further improvement
		if (r == c1)
			break;              // Break point rarely
		s = ( r - c1 );             // (r-1)
		s.adjustExponent(-1);       // 0.5*(r-1)
		if (first_time == true)
			{
			// pow3=3^0=1 already initialized
			first_time = false;
			}
		else
			pow3 *= c3;         // 3^(n-1)
		rsq = r.square();
		a = a * rsq - pow3 *(rsq - c1);
```

```
        }

    pi = a.inverse();                   // pi = 1/a
    pi.precision(digits);               // Round to precision
    return pi;
    }
```

Borwein Quintic (fifth order) Algorithm

As Borwein researched more, they found higher order convergence algorithm for π.

This is the quantic algorithm where the number of correct digits quintuples after each iteration.

Let the initial conditions be: $a_0 = \frac{1}{2},\ s_0 = 5(\sqrt{5} - 2)$

Then for n=1,2,3…

$$x_n = \frac{5}{s_{n-1}} - 1$$

$$y_n = (x_n - 1)^2 + 7$$

$$z_n = \sqrt[5]{\frac{1}{2}x_n(y_n + \sqrt{y_n^2 - 4x_n^3})}$$

$$a_n = s_{n-1}^2 a_{n-1} - 5^{n-1}(\frac{s_{n-1}^2 - 5}{2} + \sqrt{s_{n-1}(s_{n-1}^2 - 2s_{n-1} + 5)})$$

$$s_n = \frac{25}{(z_n + \frac{x_n}{z_n} + 1)^2 s_{n-1}}$$

$$\pi_n = \frac{1}{a_n}$$

As you can see, it requires only two iterations to get the result.

π

Iteration	x	y	z	a	s	π	error
0				0.5	1.18034	n	
1	3.236068	12	1.890039	0.318316	1.000001	3.14153694751872	5.57E-05
2	3.999997	15.99998	2	0.31831	1	3.14159265358980	4.44E-15

Algorithm 3.7 Borwein quantic (5th order convergence)

```
// Borwein quintic algorithm
float_precision pi_borwein_quintic(unsigned int digits)
	{
	const size_t precision = digits + 2 + (size_t)(log10(digits) + 0.5);
	bool first_time = true;
	const float_precision c0(0), c1(1), c2(2), c05(0.5), c4(4), c5(5), c7(7);
	const float_precision c16(16), c25(25);
	float_precision a(0.5, precision), an(0, precision), s(5, precision);
	float_precision x(0, precision), y(0, precision), z(0, precision);
	float_precision pow5(1, precision), ssq(0, precision), tmp(0,precision);
	float_precision pi(0, precision);
	const eptype limit = -(eptype)ceil((digits + 2)*log2(10));
	size_t loopcnt = 1;

	s = c5*(sqrt(s) - c2);                    // 0.5*(sqrt(5)-1)
	for(; ;++loopcnt)
		{
		// Build x
		x = c5 / s - c1;                  // 5/s-1
		// Build y
		y = x - c1; y *= y; y += c7;      // (x-1)^2+7
		// Build z
		z = y.square() - c4*x*x.square(); // y^2-4x^3
		// due to potential small rounding errors you could get a very small
negative number
		// if z<0 then set z=0 and continue
		if (z < c0)
			z = c0;
		else
			z = sqrt(z);
		z = x*(y + z);                    // x*(y+sqrt(y^2-4x^3))
		z.adjustExponent(-1);             // 0.5*x*(y+sqrt(y^2-4x^3))
		z = nroot(z, 5);                  // ( 0.5*x*(y+sqrt(y^2-
4x^3)))^(1/5)
		// Build an
		ssq = s.square();                 // s*s
		tmp = s*(ssq - c2 * s + c5);      // s^*(s^2-2*s+5)
		tmp = sqrt(tmp);                  // sqrt(s^*(s^2-2*s+5))
		tmp += c05*(ssq - c5);            // sqrt(s^*(s^2-2*s+5))+0.5(s^2-5)
		if (first_time == true)
			{
			//pow5 = c1;                // 5^0=1
			first_time = false;
			}
		else
			pow5 *= c5;               // 5^n
		tmp *= pow5;
		an = ssq*a - tmp;
		// Build s
		tmp = z + x / z + c1;             // z+x/z+1
		tmp = tmp.square() * s;           // s(z+x/z+1)^2
		tmp = c25 / tmp;                  // 25/(s(z+x/z+1)^2)
		if ((tmp - s).exponent() < limit)
			break;
		s = tmp;                          // 25/(s(z+x/z+1)^2)
		if (s == c1)
```

```
                break;
            a = an;
            }

        pi = an.inverse();                  // pi = 1/a
        pi.precision(digits);               // Round to precision
        return pi;
        }
```

Borwein Nonic (ninth order) algorithm

This is the Nonic convergence algorithm where the number of correct digits is 9 times more after each iteration.

Let the initial conditions be: $a_0 = \frac{1}{3},\ r_0 = \frac{\sqrt{3}-1}{2}, s_0 = \sqrt[3]{1 - r_0^3}$

Then for n=1,2,3…

$$t_n = 1 + 2r_{n-1}$$

$$u_n = \sqrt[3]{9r_{n-1}(1 + r_{n-1} + r_{n-1}^2)}$$

$$v_n = t_n^2 + t_n u_n + u_n^2$$

$$w_n = \frac{27(1 + s_{n-1} + s_{n-1}^2)}{v_n}$$

$$a_n = w_n a_{n-1} + 3^{2n-3}(1 - w_n)$$

$$s_n = \frac{(1 - r_{n-1})^3}{(t_n + 2u_n)v_n}$$

$$r_n = \sqrt[3]{1 - s_n^3}$$

$$\pi_n = \frac{1}{a_n}$$

As you can see, we use only two iterations to get the best result possible using approximately 15 digits of precision in our calculation.

π

Iteration	T	u	v	w	a	s	r	π	error
0					0.33333	0.9834	0.3660		
1	1.7320	1.7033	8.8512	9	0.33333	0.0056	1	3.00000000000000	0.141593
2	3	3	27	1.0056	0.31831	8.3E-25	1	3.14159265358980	5.77E-15

It should be noted although the Quintic and the Nonic algorithm has considerably higher convergence rate than the Quadratic and Cubic then the extra work of calculation is not worth the fewer iteration needed.

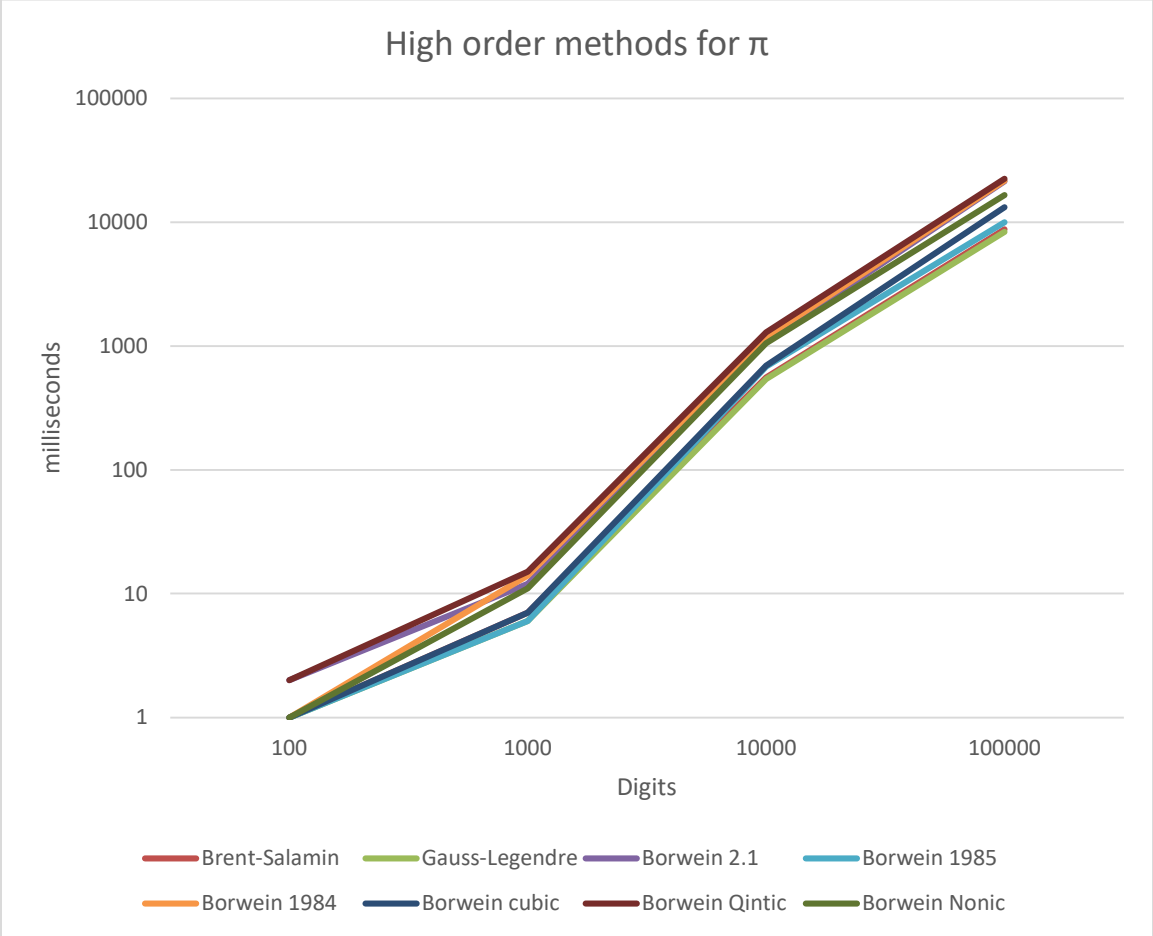
Algorithm 3.8 Borwein Nonic (9th order convergence)

```
// Borwein nonic order algorithm
float_precision pi_borwein_nonic(unsigned int digits)
	{
	const size_t precision = digits + 2 + (size_t)(log10(digits) + 0.5);
	bool first_time = true;
	const float_precision c1(1), c2(2), c3(3), c05(0.5), c9(9), c27(27);
	float_precision a(3, precision), an(0, precision), s(5, precision);
	float_precision u(0, precision), t(0, precision), r(3, precision);
	float_precision pow3(0, precision), v(0, precision), w(0, precision);
	float_precision pi(0,precision);
	const eptype limit = -(eptype)ceil((digits + 2)*log2(10));
	size_t loopcnt = 1;

	a = a.inverse();                    // 1/3
	r = c05 * ( sqrt(r) - c1);          // (sqrt(3)-1)/2
	s = nroot(c1-r*r*r,3);              // (1-r)^(1/3)
	for (; ;++loopcnt)
		{
		// Build t
		t = c1 + c2*r;              // 1+2r
		// Build u
		u = c1 + r + r * r;         // 1+r+r^2
		u *= c9 * r;                // 9r(1+r+r^2)
		u = nroot(u, 3);            // (9r(1+r+r^2))^(1/3)
		// Build v
		v = t*t + t*u + u*u;        // t^2+tu+u^2
		// Build w
		w = c27*(c1 + s + s*s);     // 27(1+s+s^2)
		w /= v;                     // 27(1+s+s^2)/v
		// Build an
		if (first_time == true)
			{
			pow3 = a;           // 3^(-1)=1/3 same as the initial a
			first_time = false;
			}
		else
			pow3 *= c9;         // 3^(2n-1)
		an = w*a+pow3*(c1 - w);     // w*a+3^(2n-1)(1-w)
		// Build s
		s = (c1 - r);               // (1-r)
		s *= s.square();            // (1-r)^3
		s /= (t + 2 * u)*v;         // (1-r)^3/((t+2u)*v)
		// Build  r
		r = c1 - s.square() * s;    // 1-s^3
		r = nroot(r, 3);            // (1-s^3)^(1/3)
		pi = an - a;
		if (loopcnt>1 && (r==c1||an == a||(pi).exponent()<limit))
			break;
		a = an;
		}

	pi = an.inverse();          // pi = 1/a
	pi.precision(digits);       // Round to precision
```

```
        return pi;
        }
```

Performance of higher order methods for π

The performance of the higher-order methods is listed below. It is not a surprise to see that higher-order convergence methods like Borwein Quintic and Nonic version does not provide better performance compared to quadratic convergence methods. Usually, the promise of these higher-order methods is eaten up by the much higher complexity of each iteration step. A clear winner is the Brent-Salamin & Gauss-Legendre methods (very similar) which are approx. 2.5 times faster than Borwein 1984 methods and approx. 2-2.5 times faster than Borwein Quintic and Nonic methods.



Recommendation for higher order π

Although all the higher-order methods are within a factor of three from top to bottom. My recommendation will be to use the Brent-Salamin method. Higher order methods like

Borwein 9^{th} order or 5^{th} order although faster convergence rate the methods complexity is higher than the gain in performance and is therefore not recommended.

Spigot Algorithm

The spigot algorithm for calculating π was discovered by Rabinowitz-Wagon in 1990. See [12]. The formula is remarkably simple and does not require any fancy computing just the use of basic operations like, add, subtract, multiply, and divide.
However, it still requires that to compute the n-digits of π, you still need to calculate all the preceding n-1 digits. Although invented solely by Plouffe in 1995, the paper from Bailey, Borwein, and Plouffe describes the methods in detail. [11].
The algorithm goes as follows:

$$\pi = \sum_{n=0}^{\infty} \frac{1}{16^n}\left(\frac{4}{8n+1} - \frac{2}{8n+4} - \frac{1}{8n+5} - \frac{1}{8n+6}\right)$$

This algorithm is not faster than the higher-order algorithm just describe but what is interesting here is that it does not require more than just basic operators. It takes nine iterations to get to approx. 15 digits accuracy.

Iteration	Σ	π	Error
0	3.133333	3.13333333333333	8.26E-03
1	8.09E-03	3.14142246642247	1.70E-04
2	1.65E-04	3.14158739034658	5.26E-06
3	5.07E-06	3.14159245756744	1.96E-07
4	1.88E-07	3.14159264546034	8.13E-09
5	7.77E-09	3.14159265322809	3.62E-10
6	3.45E-10	3.14159265357288	1.69E-11
7	1.61E-11	3.14159265358897	8.20E-13
8	7.80E-13	3.14159265358975	4.09E-14
9	3.89E-14	3.14159265358979	1.78E-15

Not very impressive but it was also discovered that this algorithm could be used so you can calculate the next digit of π without knowledge of all the preceding values of π digits. This is in sharp contrast to the higher order methods where you are bound to perform the iteration with the number of digits you need to calculate π. Even if you need considerably fewer iterations to calculate π for the higher orders methods e.g. 1 Million digits of π can be done in approx. 13 iterations for cubic convergence methods then you still have to perform all the calculations using 1 million digits which even with speedup methods can be slow.

The spigot algorithm is based on the expansion for π:

Practical implementation of π Algorithms

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

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

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

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

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

This series expands out using the Horner schema:

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

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

You can set up a simple excel sheet calculating the digits in π as the one below, see [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 Algorithm

	π		1 3	2 5	3 7	4 9	5 11	6 13	7 15	8 17	9 19	10 21	11 23	12 25	13 27	14 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	90	
Sum		13	40	53	80	95	148	108	218	201	180	365	356	330	290	200

remainders		3	1	3	3	5	5	4	8	14	9	8	11	5	20	26
Scale		30	10	30	30	50	50	40	80	140	90	80	110	50	200	260
Carry	**4**	11	24	30	40	45	54	77	96	72	80	88	84	143	120	
Sum		41	34	60	70	95	104	117	176	212	170	168	194	193	320	260

remainders		1	1	0	0	5	5	0	11	8	18	0	10	18	23	28
Scale		10	10	0	0	50	50	0	110	80	180	0	100	180	230	280
Carry	**1**	5	6	15	40	40	42	91	88	108	50	121	156	169	135	
Sum		15	16	15	40	90	92	91	198	188	230	121	256	349	365	280

remainders		5	1	0	5	0	4	0	3	1	2	16	3	24	14	19
Scaler		50	10	0	50	0	40	0	30	10	20	160	30	240	140	190
Carry	**5**	6	8	24	8	20	6	21	24	54	110	88	156	104	90	
Sum		56	18	24	58	20	46	21	54	64	130	248	186	344	230	190

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

Algorithm 4.1 Gibbons spigot

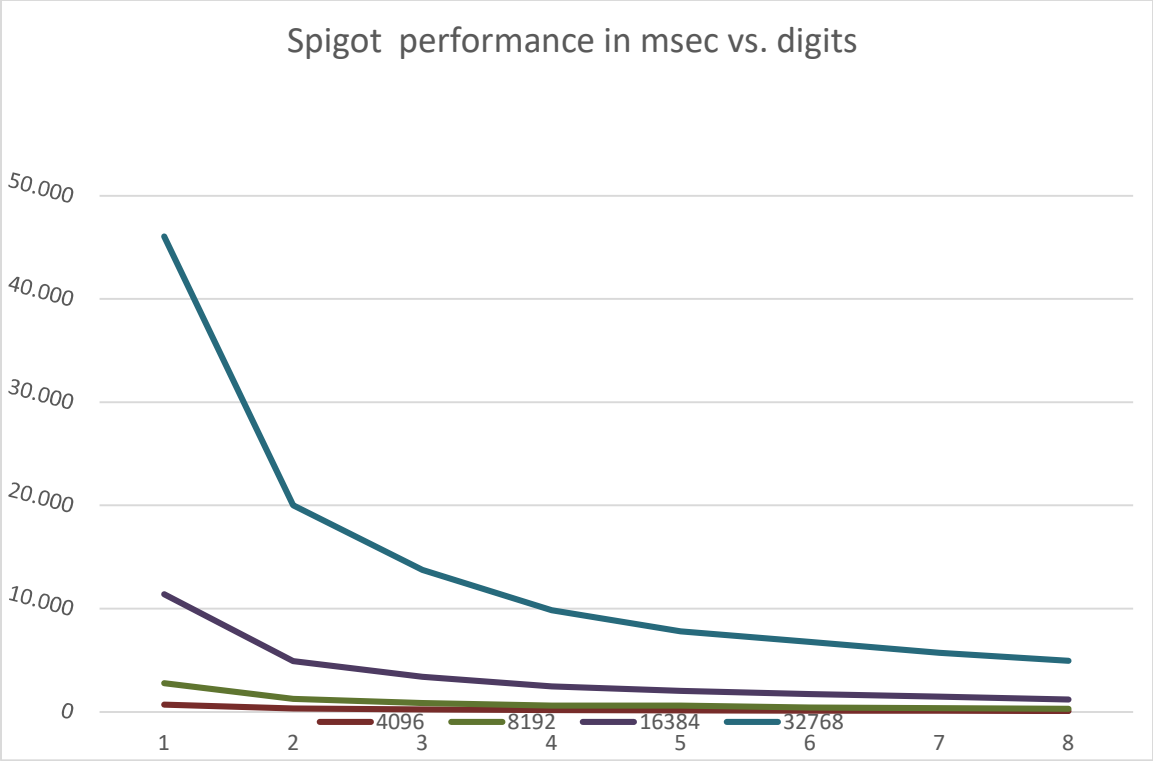
```
#define NDIGITS  15000             /*max.digits to compute*/
#define LEN      (NDIGITS/4+1)*14 /*nec.arraylength*/
Int a[LEN];                        /*arrayof4digit-decimals*/
Int b;                             /*nominatorprev.base*/
Int c=LEN;                         /*index*/
Int d=0;                           /*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 in overflow. Let us try to improve that and make it more useful to generate more numbers of π digits. The first improvement we can do is to use 32-bit unsigned integer arithmetic instead of signed integer arithmetic which 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 on the magnitude of $2*10^7$. The maximum digit that can be held using 32-bit unsigned integer arithmetic is $\sim 4*10^9$ and that is why the algorithm goes into overflow shortly after 30,000 digits of π have been calculated.

The next thing we can do is to lower the number of digits to add for each iteration, 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. Continuing down that road we can increase the digits for PI to ~2,8 million digits and more than 26 million digits if we only find one digit at a time. However, this is not without a time penalty. See the picture below that shows 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 the below algorithm that only uses 32-bit integer arithmetic. The maximum number of digits it can handle is five digits at a time but that will limit the digits of π to 3,474 before it goes into overflow.



Gibbon's 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. And 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 generates 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 64-bit version of the final algorithm is listed below.

Algorithm 4.2 64-bit Spigot

```
// 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 do nearly "unlimited"
// By default it collects 4 digits at a time, parameter no_dig can adjust that from 1 to 8
std::string pi_spigot_64( const int digits, int no_dig = 4 )
        {
        static const unsigned long f_table[]=
{0,10,100,1000,10000,100000,1000000,10000000,100000000};
        static const unsigned long f2_table[] = {0,2,20,200,2000,20000,200000,2000000,20000000};
        const int TERMS = (10 * no_dig / 3 + 1);          // Number of Terms needed
        bool first_time = true;                           // First time in loop flag
```

```
        bool overflow_flag = false;                     // Overflow flag
        char buffer[9];
        std::string ss;                 // The String that holds the calculated PI
        long b, c;                      // Loop counters
        int carry, no_carry = 0; // Outer loop carrier, plus no of carrier 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 long acc = 0, g = 0, tmp64;
        ss.reserve(digits + 16);        // 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 are divisible 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 divisible by
no_dig
        f = f_table[no_dig];            // Load the initial f
        f2 = f2_table[no_dig];          // Load the initial f2
        std::vector<uintmax_t> a(c);    // Vector of 8 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 the 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
occasionally be no_dig+1 digits in which case we have to propagate back the extra digit.
                // Check for extra carry to propagate back into the current sum of PI digits
                carry = (unsigned)( dig_n / f);
                // Eliminate the extra carrier. Now l contains no_dig digits to add to the string
                dig_n %= f;
                // Add the carrier to the existing number for PI calculate so far.
                if (carry > 0)
                        {
                        ++no_carry;
                        // Keep count of how many carriers detect
                        for (size_t i = ss.length(); carry > 0 && i > 0; --i)
                        // Loop and propagate back the extra carrier to the existing
                        // PI digits found so far
                                {
                                // It can handle multiple carry-back propagations
                                int new_digit;
                                new_digit = (ss[i - 1] - '0') + carry;   // Calculate new digit
                                carry = new_digit / 10;  // Calculate new carry if any
                                // Put the adjusted digit back in our PI digit list
                                ss[i - 1] = new_digit % 10 + '0';
                                }
                        }

                for(int i = no_dig-1; i >= 0; --i)
                        {
```

```
                        if (dig_n > 0)
                                {
                                buffer[i] = (dig_n % 10)+'0'; dig_n /= 10;
                                }
                        else
                                buffer[i] = '0';
                        }
                // Print previous no_dig digits to buffer
                ss += std::string(buffer,no_dig);
                // Add it to PI string
                if(first_time==true)
                        ss.insert(1, ".");
                // add the decimal point after the first digit to create 3.14...
                acc = acc % f;
                // save current no_dig digits and repeat loop
                e = (unsigned long)acc;
                }
        // Remove the extra digits that we didn't request but used as guard digits
        ss.erase(digits+1);
        if (overflow_flag == true)
                ss = std::string("Overflow:") + ss;
        // Set overflow in the return string
        return ss;      // Return Pi with the number of digits
        }
```

Gosper formula for π

As it has been mentioned in the previous section, Rabinowitz-Wagon Spigot Algorithms for π require approximately 3.3 terms per digit of π. However, Gosper page formula for π can also be used and it is more efficient since it requires fewer terms to be evaluated per digit. 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 digit, however as always you do not get things for free. Each term is a little bit more complicated to handle and you quickly reach the limit of the integer representation so for all practical purposes 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 expands into this series:

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

And:

Practical implementation of π Algorithms

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

This is the way we want to have the series expanded so we can quickly identify the different Spigot elements. This is a 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 of 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 is the one below, see [17] 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. In the below table, we see that we only need six terms to get approximately seven correct digits which is a lot less that the Rabinowitz-Wagon algorithm. However, you also notice that the mixed radix-based $\frac{A}{B}$ quickly gets 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). The only drawback with the Gosper algorithm over the Rabinowitz-Wagon Spigot Algorithms for π is that the mixed radix based $\left(\frac{1}{60}, \frac{6}{168}, \frac{15}{330}, \frac{28}{816}, \ldots, \frac{n(2n-1)}{3(9(n^2+n)+2)}, \ldots\right)$ yield higher than the Rabinowitz-Wagon algorithm that used $\left(\frac{1}{3}, \frac{2}{5}, \frac{3}{7}, \frac{4}{9}, \ldots, \frac{n}{2n+1}\right)$. This leads to faster overflow even when using 64-bit integer arithmetic. On the other hand, we do not need as many terms as the Rabinowitz-Wagon Algorithm. For a wanted precision of *d* digits, we need the Rabinowitz-Wagon algorithm n=$(\frac{10d}{3} + 1)$. For the Gosper algorithm, we need $n \sim \frac{d}{0.9}$. Dividing the two formulas you get a ratio of $\sim 3 + \frac{0.9}{d}$=> or 3 for a larger number of d. e.g let's assume you need to find 1,000,000 digits precision of π. The largest term needs for Gosper even with the reduced number of terms is $\frac{\sim 6.67^{11}}{\sim 9^{12}}$ while Rabinowitz-wagon is $\frac{10^6}{\sim 2x10^6}$. We need to expect the Gosper algorithm to overflow faster for large *d* than the Rabinowitz-Wagon algorithm.

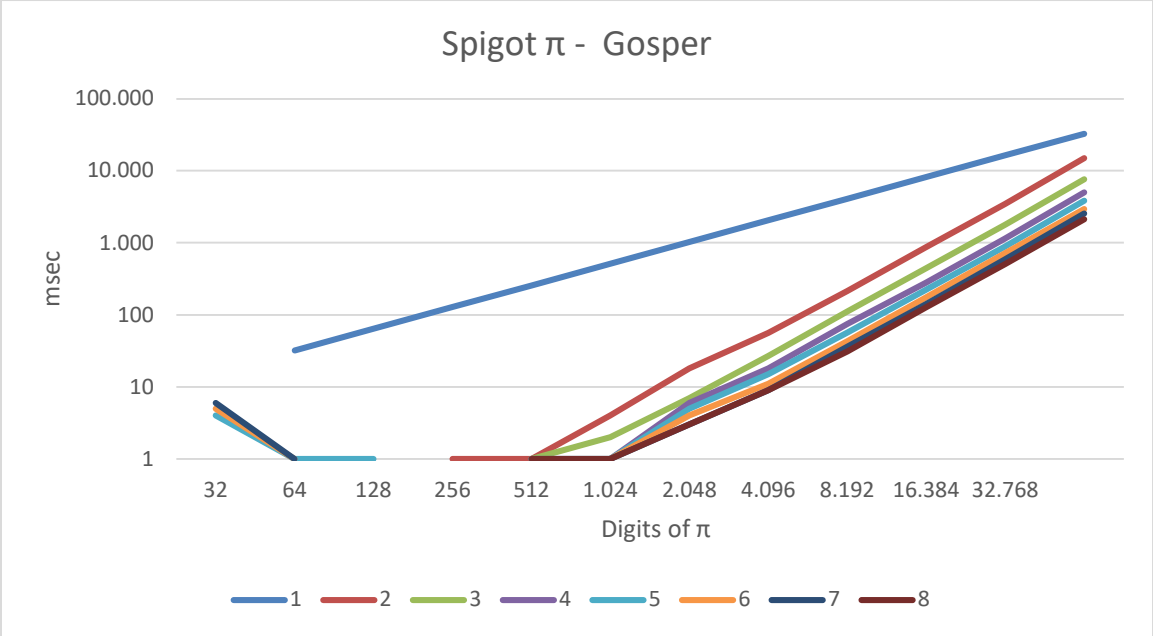
Algorithm 4.3 Gosper 64-bit

```
// 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[9];
        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 are
divisible 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
        std::vector<uintmax_t> acc(no_terms+1);// Vector of the 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';
                                }
                        }
```

```
            for (int i = no_dig - 1; i >= 0; --i)
                    {
                    if (dig_n > 0)
                            {
                            buffer[i] = (dig_n % 10) + '0'; dig_n /= 10;
                            }
                    else
                            buffer[i] = '0';
                    }
            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;
    return ss;
    }
```

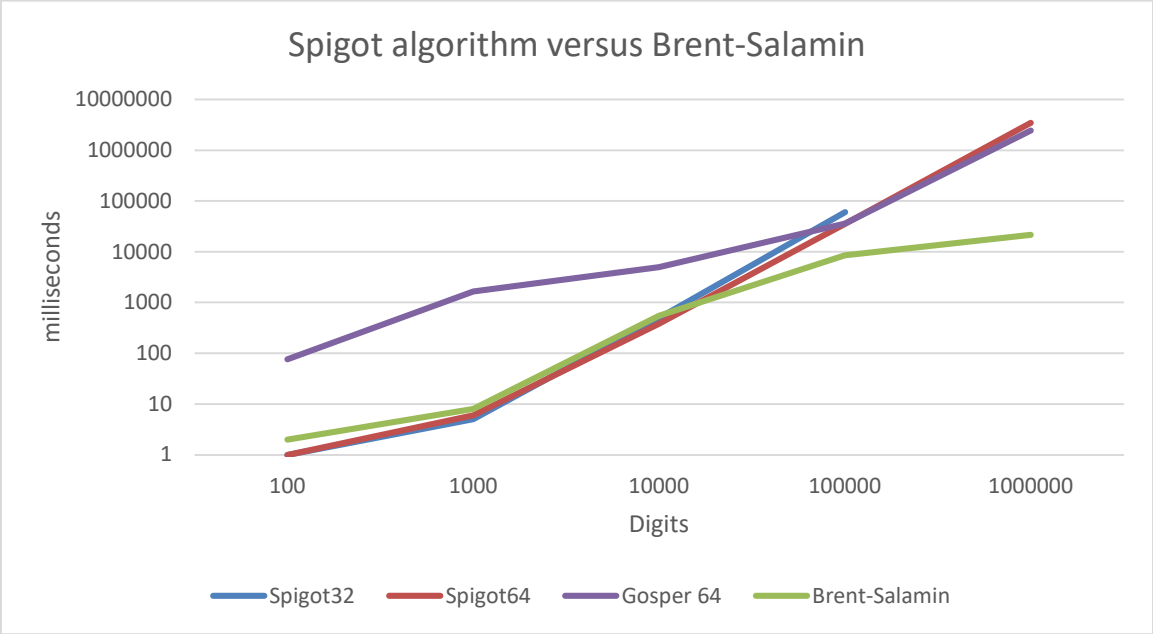
The above mention algorithm can find π and for each loop, we can find between one and eight digits. Not surprisingly, the more digits we find per loop the faster the overall algorithm is as shown in the below diagram. The numbers 1 to 8 refer to how many digits we find per loop and in the calculation, 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 than applying the algorithm one digit at a time

Time comparison is that the 32-bit is faster in the range of π digits that both algorithms can handle. This is not a surprise since 64-bit integer arithmetic is more time-consuming that the equivalent 32-bit integer arithmetic. However, as the number of digits increases above 1,000digits the Spigot algorithm is lacking that of the Brent-Salamin algorithm. It

is interesting to note that the 64-bit spigot is faster than the Gosper version for digits up to 100,000, whereas the Gosper algorithm has faster performance.



Recommendation for Spigot-π algorithms.

- Always use the 64-bit version since it allows you to do 8 digits at a time, which is faster than the 32-bit version.
- Consider the Gosper version when the number of digits exceeds 100,000.

Recommendation for π algorithms.

- The Chudnovsky binary splitting method is the fastest of them all. See the chart below. It is no surprise that this algorithm is also being used to break the records for the calculation of many digits for π. The Brent-Salamin method is fast but the binary splitting method is in a league of its own.

- If the Binary splitting method is not an option then I recommend the Brent-Salamin versions.
- If there is no arbitrary precision library available then the Spigot 64-bit or Gosper 64-bit can be useful.

Reference

1. Arbitrary precision library package. Arbitrary Precision C++ Packages (hvks.com)
2. The World of π. www.pi314.net/eng/salamin.php - Oct 5, 2016
3. The Math Forum: http://mathforum.org/library/drmath/view/58283.html - Oct 5, 2016
4. Borwein, Pi and the AGM, Volume 4, John Willey & Sons Inc, New York, NY 1998
5. https://en.wikipedia.org/wiki/Borwein%27s_algorithm – Oct 6, 2016
6. https://en.wikipedia.org/wiki/Bailey%E2%80%93Borwein%E2%80%93Plouffe_formula – Oct 6, 2016
7. https://en.wikipedia.org/wiki/Approximations_of_%CF%80 – Oct 6, 2016
8. P. Borwein – The Amazing number π
9. Bailey, Borwein, Plouffe, The Quest for Pi, June 25, 1996
10. J. Borwein, Ramanujan and PI, May 3 2012
11. J Borwein, The life of Pi: From Archimedes to Eniac and Beyond, June 19, 2012
12. Bailey, Borwein, Plouffe, "on the rapid Computation of Various Polylogarithmic Constants 1996. http://www.cecm.sfu/personal/pborwein
13. Rabinowitz & Wagon, A Spigot Algorithm for the Digits of Pi, The American Mathematical Monthly, 102 (1995) page 195-203.
14. Unbounded Spigot Algorithms for the Digits of PI
15. Binary Splitting Recursion Library (numberworld.org)
16. The world of π. http://www.pi314.net/eng/goutte.php - Dec 28, 2016

Appendix

The binary splitting method for Ramanujan, Chudnovsky, and the Borwein π is outlined below.

The Binary Recursion algorithm for the three-variable splitting

The three-variable binary splitting recursion goes as follows [15]:

$$x = \sum_{k=1}^{n} \frac{P(k)}{R(k)} \prod_{i=1}^{k} \frac{R(i)}{Q(i)} = \frac{P(0,n)}{Q(0,n)}$$

Given a<m<b.

$m = \frac{a+b}{2}\ integer\ division$
P(a,b)=P(a,m)Q(m,b)+P(m,b)R(a,m)
Q(a,b)=Q(a,m)Q(m,b)
R(a,b)=R(a,m)R(m,b)

Where:
P(b+1,b)=P(b)
Q(b-1,b)=Q(b)
R(b-1,b)=R(b)

Deriving the Binary splitting method for Ramanujan π

The binary splitting algorithm is on the form:

$$x = \sum_{k=1}^{n} \frac{P(k)}{R(k)} \prod_{i=1}^{k} \frac{R(i)}{Q(i)}$$

In addition, we need to get the Ramanujan π series into that form.
Starting with the Ramanujan series for π:

$$\frac{1}{\pi} = \frac{2\sqrt{2}}{9801} \sum_{k=0}^{\infty} \frac{(4k)!\,(1103 + 26390k)}{(k!)^4 396^{4k}}$$

The first step is to split up the factorial into product notation from the rest of the series.

Noting (4k)! in product notation is: $\prod_{i=1}^{k}(4i)(4i-1)(4i-2)(4i-3)$ and
$(k!)^4$ is: $\prod_{i=1}^{k} i^4$

$$\frac{1}{\pi}=\frac{2\sqrt{2}}{9801}\sum_{k=0}^{\infty}\frac{1103+26390k}{396^{4k}}\prod_{i=1}^{k}\frac{(4i)(4i-1)(4i-2)(4i-3)}{i^4}$$

Aligning the start index to one and moving out the constant at k=0 yields:

$$\frac{1}{\pi}=\frac{1103\sqrt{2}}{9801}+\frac{2\sqrt{2}}{9801}\sum_{k=1}^{\infty}\frac{1103+26390k}{396^{4k}}\prod_{i=1}^{k}\frac{(4i)(4i-1)(4i-2)(4i-3)}{i^4}$$

Simplifying the product notation part:

$$\frac{1}{\pi}=\frac{1103\sqrt{2}}{9801}+\frac{2\sqrt{2}}{9801}\sum_{k=1}^{\infty}\frac{1103+26390k}{396^{4k}}\prod_{i=1}^{k}\frac{(4i-1)(2i-1)(4i-3)}{\frac{1}{8}i^3}$$

Rearranging the formula to fit into the binary splitting form:

$$\frac{1}{\pi}=\frac{1103\sqrt{2}}{9801}+\frac{2\sqrt{2}}{9801}\sum_{k=1}^{\infty}\frac{(4k-1)(2k-1)(4k-3)(1103+26390k)}{(4k-1)(2k-1)(4k-3)396^{4k}}\prod_{i=1}^{k}\frac{(396^4)(4i-1)(2i-1)(4i-3)}{\frac{1}{8}(396^4)i^3}$$

Simplify:

$$\frac{1}{\pi}=\frac{1103\sqrt{2}}{9801}+\frac{2\sqrt{2}}{9801}\sum_{k=1}^{\infty}\frac{(4k-1)(2k-1)(4k-3)(1103+26390k)}{(4k-1)(2k-1)(4k-3)}\prod_{i=1}^{k}\frac{(4i-1)(2i-1)(4i-3)}{3073907232i^3}$$

Now identify the P(k), Q(k) and R(K) you get:

$P(k)=(1103+26390k)(4k-1)(2k-1)(4k-3)$
$Q(k)=3073907232k^3$
$R(k)=(4k-1)(2k-1)(4k-3)=32k^3-48k^2+22k-3$

Replacing the series with P(0,k) and Q(0,k) yields:

$$\frac{1}{\pi}=\frac{1103\sqrt{2}}{9801}+\frac{2\sqrt{2}}{9801}\frac{P(0,k)}{Q(0,k)}=>$$

$$\pi=\frac{9801Q(0,k)}{P(0,k)+11033Q(0,k)}\frac{1}{2\sqrt{2}}$$

Which is the Binary Splitting algorithm for Ramanujan π.
The linear convergent cost, which is an expression of the computational speed can be found using the formula in [15]:

$$Relative\ cost = \frac{4D}{\log\left(\frac{C_q}{C_r}\right)}$$

And you get: $\frac{4 \cdot 3}{\log\left(\frac{3073907232}{32}\right)} = 0.653$

Deriving the Binary splitting method for Chudnovsky π

Chudnovsky series for π:

$$\frac{1}{\pi} = 12\sum_{k=0}^{\infty}\frac{(-1)^k(6k)!\,(13591409 + 545140134k)}{(3k)!\,(k!)^3 640320^{3k+3/2}} =>$$

$$\frac{1}{\pi} = \frac{\sqrt{10005}}{4270934400}\sum_{k=0}^{\infty}\frac{(-1)^k(6k)!\,(13591409 + 545140134k)}{(3k)!\,(k!)^3 640320^{3k}} =>$$

Separate the factorial into product notation:
Where (6k)! in product notation is: $\prod_{i=1}^{k} 6i(6i-1)(6i-2)(6i-3)(6i-4)\,(6i-5)$
(3k)! is: $\prod_{i=1}^{k}(3i)(3i-1)(3i-2)$
And (k!)^3 is: $\prod_{i=1}^{k} i^3$

$$\frac{1}{\pi} = \frac{\sqrt{10005}}{4270934400}\sum_{k=0}^{\infty}\frac{(-1)^k(13591409 + 545140134k)}{640320^{3k}}\prod_{i=1}^{k}\frac{6i(6i-1)(6i-2)(6i-3)(6i-4)(6i-5)}{3i(3i-1)(3i-2)i^3}$$

Aligning index start and simplify you get:

$$\frac{1}{\pi} = \frac{13591409\sqrt{10005}}{4270934400} + \frac{\sqrt{10005}}{4270934400}\sum_{k=1}^{\infty}\frac{(-1)^k(13591409 + 545140134k)}{640320^{3k}}\prod_{i=1}^{k}\frac{24(6i-1)(2i-1)(6i-5)}{i^3}$$

Rearranging the formula to fit into the binary splitting form:

$$\frac{1}{\pi} = \frac{13591409\sqrt{10005}}{4270934400} + \frac{\sqrt{10005}}{4270934400}\sum_{k=1}^{\infty}\frac{(-1)^k(6k-1)(2k-1))(6k-5)(13591409 + 545140134k)}{(6k-1)(2k-1))(6k-5)640320^{3k}}\prod_{i=1}^{k}\frac{640320^3 \cdot 24(6i-1)(2i-1)(6i-5)}{640320^3 \cdot i^3}$$

Simplify:

$$\frac{1}{\pi} = \frac{13591409\sqrt{10005}}{4270934400} + \frac{\sqrt{10005}}{4270934400}\sum_{k=1}^{\infty}\frac{(-1)^k(6k-1)(2k-1))(6k-5)(13591409+545140134k)}{(6k-1)(2k-1)(6k-5)}\prod_{i=1}^{k}\frac{(6i-1)(2i-1)(6i-5)}{10939058860032000\cdot i^3}$$

Now identify the P(k), Q(k) and R(K) you get:

$$P(k) = (-1)^k(13591409+545140134k)(6k-1)(2k-1)(6k-5)$$
$$Q(k) = 10939058860032000k^3$$
$$R(k) = (6k-1)(2k-1)(6k-5) = 72k^3 - 108k^2 + 46k - 5$$

Replacing the series with P(0,k) and Q(0,k) yields:

$$\frac{1}{\pi} = \frac{13591409\sqrt{10005}}{4270934400} + \frac{\sqrt{10005}}{4270934400}\frac{P(0,k)}{Q(0,k)} =>$$

$$\pi = \frac{4270934400Q(0,k)}{P(0,k)+13591409Q(0,k)}\frac{1}{\sqrt{10005}}$$

Which is the Binary Splitting algorithm for Chudnovsky π.

The relative cost is $Relative\ cost = \frac{4D}{\log(\frac{Cq}{Cr})} = \frac{4\cdot 3}{\log(\frac{10939058860032000}{72})} = 0.367$

Deriving the Binary splitting method for Borwein25 π

The Binary Splitting algorithm for Borwein25 for π.

Borwein 25 series for π:

$$\frac{1}{\pi} = 12\sum_{k=0}^{\infty}\frac{(-1)^k(6k)!\,(A+kB)}{(3k)!\,(k!)^3C^{k+1/2}} =>$$

$$\frac{1}{\pi} = \frac{12}{\sqrt{C}}\sum_{k=0}^{\infty}\frac{(-1)^k(6k)!\,(A+kB)}{(3k)!\,(k!)^3C^{k}} =>$$

Separate the factorial into product notation and move the constant a k=0 out of the summation:
Where (6k)! in product notation is: $\prod_{i=1}^{k} 6i(6i-1)(6i-2)(6i-3)(6i-4)\,(6i-5)$

(3k)! is: $\prod_{i=1}^{k}(3i)(3i-1)(3i-2)$
And (k!)^3 is: $\prod_{i=1}^{k} i^3$ yields:

$$\frac{1}{\pi}=\frac{12A}{\sqrt{C}}+\frac{12}{\sqrt{C}}\sum_{k=1}^{\infty}\frac{(-1)^k(A+kB)}{C^k}\prod_{i=k}^{k}\frac{6i(6i-1)(6i-2)(6i-3)(6i-4)(6i-5)}{3i(3i-1)(3i-2)i^3}$$

Simplify:

$$\frac{1}{\pi}=\frac{12A}{\sqrt{C}}+\frac{12}{\sqrt{C}}\sum_{k=1}^{\infty}\frac{(-1)^k(A+kB)}{C^k}\prod_{i=k}^{k}\frac{24(6i-1)(2i-1)(6i-5)}{i^3}$$

Rearranging the formula to fit into the binary splitting form:

$$\frac{1}{\pi}=\frac{12A}{\sqrt{C}}+\frac{12}{\sqrt{C}}\sum_{k=1}^{\infty}\frac{(-1)^k(6i-1)(2i-1)(6i-5)(A+kB)}{(6i-1)(2i-1)(6i-5)C^k}\prod_{i=k}^{k}\frac{C\cdot(6i-1)(2i-1)(6i-5)}{\frac{C}{24}\cdot i^3}$$

Simplify:

$$\frac{1}{\pi}=\frac{12A}{\sqrt{C}}+\frac{12}{\sqrt{C}}\sum_{k=1}^{\infty}\frac{(-1)^k(6i-1)(2i-1)(6i-5)(A+kB)}{(6i-1)(2i-1)(6i-5)}\prod_{i=k}^{k}\frac{(6i-1)(2i-1)(6i-5)}{\frac{C}{24}\cdot i^3}$$

Now identify the P(k), Q(k) and R(K) you get:

$P(k)=(-1)^k(A+Bk)(6k-1)(2k-1)(6k-5)$
$Q(k)=\frac{C}{24}k^3$
$R(k)=(6k-1)(2k-1)(6k-5)=72k^3-108k^2+46k-5$

Replacing the series with P(0,k) and Q(0,k) yields:

$$\frac{1}{\pi}=\frac{12A}{\sqrt{C}}+\frac{12}{\sqrt{C}}\frac{P(0,k)}{Q(0,k)}=>$$

$$\pi=\frac{\sqrt{C}\cdot Q(0,k)}{12\cdot(P(0,k)+A\cdot Q(0,k))}$$

Which is the Binary Splitting algorithm for Borwein 25 π.

The relative cost is $Relative\ cost=\frac{4D}{\log(\frac{Cq}{Cr})}=\frac{4\cdot 3}{\log(\frac{(159999840\sqrt{61}+124963872\ )^3}{24\cdot 72})}=0.208$

Practical implementation of π Algorithms

Deriving the Binary splitting method for Borwein50 π

Borwein series that find approx. 50 correct digits per iteration:

$$\frac{1}{\pi} = \frac{1}{\sqrt{-C^3}} \sum_{k=0}^{\infty} \frac{(6k)!\,(A+nB)}{(3k)!\,(k!)^3 C^{3k}}$$

Where the constants A, B, and C are:

$$A = 6336502831297199958542622 0 + 28337702140800884204682560 0\sqrt{5} + 384\sqrt{5}(10891728551171178200467436212395209160385660 17 + 48709290865788102250773385345416887213512550 40\sqrt{5})^{\frac{1}{2}}$$

$$B = 784991045349662721028974900 0 + 351058667826093202896560640 0\sqrt{5} + 2515968\sqrt{31101}(62602083237890016369933226544440208821 61 + 27996502730604442965772068907188251902 35\sqrt{5})^{\frac{1}{2}}$$

$$C = -214772995063512240 - 96049403338648032\sqrt{5} - 1296\sqrt{5}(10985234579463550323713318473 + 49127462536923627546073959 12\sqrt{5})^{\frac{1}{2}}$$

Separate the factorial into product notation and move the constant a k=0 out of the summation:
Where (6k)! in product notation is: $\prod_{i=1}^{k} 6i(6i-1)(6i-2)(6i-3)(6i-4)\,(6i-5)$
(3k)! is: $\prod_{i=1}^{k}(3i)(3i-1)(3i-2)$
And $(k!)^3$ is: $\prod_{i=1}^{k} i^3$ yields:

$$\frac{1}{\pi} = \frac{A}{\sqrt{-C^3}} + \frac{1}{\sqrt{-C^3}} \sum_{k=1}^{\infty} \frac{(A+kB)}{C^{3k}} \prod_{i=k}^{k} \frac{6i(6i-1)(6i-2)(6i-3)(6i-4)(6i-5)}{3i(3i-1)(3i-2)i^3}$$

Simplify:

$$\frac{1}{\pi} = \frac{A}{\sqrt{-C^3}} + \frac{1}{\sqrt{-C^3}} \sum_{k=1}^{\infty} \frac{(A+kB)}{C^{3k}} \prod_{i=k}^{k} \frac{24(6i-1)(2i-1)(6i-5)}{i^3}$$

Rearranging the formula to fit into the binary splitting form:

$$\frac{1}{\pi} = \frac{A}{\sqrt{-C^3}} + \frac{1}{\sqrt{-C^3}} \sum_{k=1}^{\infty} \frac{(6i-1)(2i-1)(6i-5)(A+kB)}{(6i-1)(2i-1)(6i-5)C^{3k}} \prod_{i=k}^{k} \frac{C^3 \cdot 24(6i-1)(2i-1)(6i-5)}{C^3 \cdot i^3}$$

Simplify:

$$\frac{1}{\pi} = \frac{A}{\sqrt{-C^3}} + \frac{1}{\sqrt{-C^3}} \sum_{k=1}^{\infty} \frac{(6i-1)(2i-1)(6i-5)(A+kB)}{(6i-1)(2i-1)(6i-5)} \prod_{i=k}^{k} \frac{(6i-1)(2i-1)(6i-5)}{\frac{C^3}{24} \cdot i^3}$$

Now identify the P(k), Q(k) and R(K) you get:

$$P(k) = (A+Bk)(6k-1)(2k-1)(6k-5)$$
$$Q(k) = \frac{C^3}{24} k^3$$
$$R(k) = (6k-1)(2k-1)(6k-5) = 72k^3 - 108k^2 + 46k - 5$$

Replacing the series with P(0,k) and Q(0,k) yields:

$$\frac{1}{\pi} = \frac{A}{\sqrt{-C^3}} + \frac{1}{\sqrt{-C^3}} \frac{P(0,k)}{Q(0,k)} =>$$

$$\pi = \frac{\sqrt{-C^3} \cdot Q(0,k)}{(P(0,k) + A \cdot Q(0,k))}$$

Which is the Binary Splitting algorithm for Borwein 50 π.

The relative cost is $Relative\ cost = \frac{4D}{\log(\frac{C_q}{C_r})} = 0.103$