Computation of the n-th decimal digit of π with low memory

Xavier Gourdon

February 11, 2003

Abstract

This paper presents an algorithm that computes directly the n-th decimal digit of π in sub-quadratic time and very low memory. It improves previous results of Simon Plouffe, later refined by Fabrice Bellard. The problem of the n-th digit computation in base 2 had already been successfully treated thanks to the use of appropriate series, but no corresponding formula for the question in base 10 has been found yet. However, our result is a progress. Another result in this paper permits to compute directly the n-th decimal digit of π with intermediate memory size, leading to intermediate time complexity between linear and quadratic.

1 Introduction

The fascination of the number π by mathematicians is ancient, and numerous computations of its digits have been performed in the history. Later, computers have been used to increase the number of computed digits. The largest computation as of today is impressive : more than 1241 trillions (1.241×10^{12}) digits of π have been recently computed on a super computer by Yasumasa Kanada and his team [6]. Home computers are far from being able to reach these sizes ; for example, the data of the latest computation could fill around one thousand of CD-roms. The largest values reached today on home computer is 12 billion digits, by Shigeru Kondo, who ran the program *pifast* [8] written by the author. To over-pass the memory limitation on home computers, *pifast* extensively makes use of disk memory, but even this possibility does not permit to reach the feat of super computers.

Recently, another angle to the problem have been considered by Bailey, Borwein and Plouffe in [1]. They found the formula

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

which permits to compute directly the n-th bit of π with memory $O(\log n)$ and quasi-linear time $O(n \log^3 n)$. Later, a better series was found by Fabrice Bellard, which improves by 43% the efficiency. Fabrice Bellard formula was used in a distributed project on Internet [5] by Colin Percival to compute successfully the quadrillionth (10^{15}) bit of π, using associated efforts of more than one thousand home computers in the world for a total time of 1.2 million cpu hours.

It is of course natural to try to get the same kind of results in base 10, but no formula of the kind of (1) have been found yet that naturally fits to the decimal digit computation. However, other techniques could be applied, and Simon Plouffe [7] was the first to propose a solution with an algorithm in time $O(n^3 \log^3 n)$ and memory $O(\log n)$, based on the formula

$$\pi + 3 = \sum_{k>0} \frac{k 2^k}{\binom{2k}{k}}.$$

This formula has no big powers of primes in denominators, and this was the key success factor in Plouffe approach. Later, based on the same formula, Fabrice Bellard refined Plouffe technique with an algorithm using $O(n^2)$ elementary operations on numbers of size $O(\log n)$. Our paper improves this result, using a completely different technique based on a series acceleration process, leading to an algorithm that permits to compute the n-th decimal digit of π using $O(n^2 \log\log n/\log^2 n)$ elementary operations on numbers of size $O(\log n)$, with memory $O(\log^2 n)$ (see theorem 1). Between very low and very large memory, there is also the question of obtained techniques with intermediate memory size and better time complexity, and we exhibit such results in theorem 2.

2 A formula suited to n-th decimal digit computation of π

Our starting point is the classical following alternating series to compute π :

$$\frac{\pi}{4} = \arctan(1) = \sum_{k=0}^{\infty} \frac{(-1)^k}{2k+1}. \tag{2}$$

In this form, the formula is well suited to n-th decimal digit computation, but its convergence is too slow. This problem is overcomed with the use of a general alternating series acceleration technique described below.

2.1 A general alternating series acceleration process

Let an alternating series of the form

$$S = \sum_{k=0}^{\infty} (-1)^k a_k$$

where we assume that there exists a *positive* function weight $w(x)$ such that

$$a_k = \int_0^1 x^k w(x)\, dx. \tag{3}$$

Various acceleration convergence techniques exist for such series (finite differences Euler acceleration process for example, ...), and can be generalized with the following result from Cohen, Villegas and Zagier [4].

Let $P_n(x) = \sum_{k=0}^{n} p_k (-x)^k$ a degree n polynomial for which $P_n(-1) \neq 0$. We define

$$S_n = \frac{1}{P_n(-1)} \sum_{k=0}^{n-1} c_k (-1)^k a_k, \qquad \text{where} \qquad c_k = \sum_{j=k+1}^{n} p_j.$$

Then we have the following bound :

$$|S_n - S| <= \frac{1}{|P_n(-1)|} \int_0^1 \frac{|P_n(x)| w(x)}{1+x}\, dx \leq \frac{M_n}{|P_n(-1)|} S \qquad \text{with} \qquad M_n = \sup_{x\in[0,1]} |P_n(x)|. \tag{4}$$

This inequality suggests to choose polynomials (P_n) with small values of $M_n/|P_n(-1)|$.

2.2 Application to the n-th decimal digit computation of π

2.2.1 Construction of the formula

This acceleration process can be applied to the alternating series (2) since the relation (3) is fulfilled with the positive weight function $w(x) = 2x^{-1/2}$. We choose the polynomial P_m in the form

$$P_m(x) = \left(x^M (1-x)\right)^N, \tag{5}$$

where, for convenience, N is even. The degree of this polynomial is $m = (M+1)N$ and it maximum value on $[0, 1]$ is obtained for $x = 1 - 1/(M+1)$, thus

$$M_m = \sup_{x\in[0,1]} |P_m(x)| = \left(\left(1 - \frac{1}{M+1}\right)^M \frac{1}{M+1}\right)^N$$
$$= \left(\left(1 - \frac{1}{M+1}\right)^{M+1} \frac{1}{M}\right)^N \leq \left(\frac{1}{eM}\right)^N,$$

where $e = \exp(1)$. Applying the bound (4), we obtain

$$|S_m - \pi| \leq \frac{\pi}{(2eM)^N}, \qquad \text{with} \qquad S_m = \sum_{k=0}^{m-1} (-1)^k \frac{c_k}{P_m(-1)} \frac{4}{2k+1} \tag{6}$$

where c_k denotes the sum of the coefficients of $P_m(-x)$ for indexes $j > k$. The polynomial P_m has its coefficients vanishing for index $j < MN$, and its particular form easily entails that our value of our approximation to π is

$$S_m = \sum_{k=0}^{(M+1)N-1} (-1)^k \frac{4}{2k+1} - \sum_{k=0}^{N-1} (-1)^k \frac{4\, s_k}{2^N(2MN + 2k + 1)}, \qquad s_k = \sum_{j=0}^{k} \binom{N}{k}. \tag{7}$$

2.2.2 Application to the n-th digit computation

Formula (7) is a good basis for a direct n-th decimal digit computation. Suppose $M \geq 2$; to compute n_0 decimal digits of π at the n-th position from the value of S_m, we need to have $|10^n S_m - 10^n \pi| < 10^{-n0}$. The bound in (6) shows that this is true as soon as

$$\frac{\pi}{(2eM)^N} < \frac{1}{10^{n+n0}}.$$

This condition will be fulfilled by choosing

$$N = \left\lceil (n + n0 + 1) \frac{\log(10)}{\log(2eM)} \right\rceil \tag{8}$$

Now, once $M \geq 4$ is fixed (and M even) and N chosen as in (8), we need to compute the fractional part of $10^n S_m$, which from (7) is equivalent to computing the fractional part of

$$\sum_{k=0}^{(M+1)N-1} (-1)^k \frac{4 \times 10^n}{2k+1} - \sum_{k=0}^{N-1} (-1)^k \frac{5^{N-2} 10^{n-N+2} s_k}{2MN + 2k + 1}.$$

The key of the success in our approach is the fact that $N \leq n + 2$, which follows from (8) as soon as n is sufficiently large compared to n_0 (in fact $n \geq 4n_0$ is sufficient), thus all numerators in the latest expression have integer value. Since the fractional part of a fraction a/b is also the fractional part of $(a \bmod b)/b$, we have proved the following proposition :

Proposition 1 *Let n be an integer, $M \geq 4$ and N defined as in (8). When $n \geq 4n_0$, we have $N \leq n$ and in this case, the fractional part of $10^n\pi$ is approximated with an error less than 10^{-n_0} by the fractional part of $B - C$, where*

$$B = \sum_{k=0}^{(M+1)N-1} (-1)^k \frac{4 \times 10^n \bmod (2k+1)}{2k+1}, \tag{9}$$

$$C = \sum_{k=0}^{N-1} (-1)^k \frac{5^{N-2} 10^{n-N+2}\, s_k \bmod (2MN + 2k + 1)}{2MN + 2k + 1}, \qquad s_k = \sum_{j=0}^{k} \binom{N}{j} \tag{10}$$

3 An algorithm to compute the n-th decimal digit with very low memory

Formulas (9) and (10) in the previous proposition permit to obtain a direct computation of the n-th decimal digit of π by using elementary operations modulo small numbers. The technique essentially consists in computing powers and sum of binomials modulo small integers and lead to a sub-quadratic algorithm (thus better than the classical quadratic algorithms to compute all the n first digits of π). More precisely, we have the following result.

Theorem 1 *Let n_0 be a fixed (small) positive integer, and $n \geq 4n_0$. Algorithm 1 below computes the fractional part of $10^n\pi$ with a precision 10^{-n_0} using $O(\log^2 n)$ memory and using $O(n^2 \log\log n/\log^2 n)$ elementary operations modulo numbers of size $O(\log n)$.*

The complexity in terms of elementary operations modulo numbers of size $O(\log n)$ is of practical interest because the implied numbers fit into 64-bits integers for reachable parameters. As for the bit complexity, interesting from a theoretical point of view, the cost of the algorithm is $O(n^2 \log\log n \times M(\log n)/\log^2 n)$ where $M(m)$ is the cost of multiplication of integers of size m. Classical multiplication corresponds to $M(m) = O(m^2)$ leading to an associated bit complexity of $O(n^2 \log\log n)$. Using Schönhage multiplication, $M(m) = O(m \log m \log\log m)$ and associated bit complexity for algorithm 1 is $O(n^2(\log\log n)^2 \log\log\log n/\log n)$ which remains sub-quadratic.

3.1 Description of the algorithm

We now detail the algorithm referenced by theorem 1.

Algorithm 1 (n-th digit computation of π with very low memory) *The following algorithm computes the fractional part of $10^n\pi$ with an error $< 10^{-n_0}$ when $n \geq 4n_0$.*

1. *Define integers M and N by*

$$M = 2\left\lceil \frac{n}{\log^3 n} \right\rceil \quad \textit{and} \quad N = \left\lceil (n + n_0 + 1)\frac{\log(10)}{\log(2eM)} \right\rceil.$$

2. *(Computation of B) Initialize $b = 0$ a floating point value. For index k, $0 \leq k < (M+1)N$ perform the following operations :*

   a *Compute $x = 4 \times 10^n \bmod 2k+1$ (classical powering modulo technique is used).*

   b *Compute $b := \{b + (-1)^k x/(2k+1)\}$.*

3. *(Computation of C) Initialize $c = 0$ a floating point value. For index k, $0 \leq k < N$ perform the following operations :*

   a *Compute $x = \sum_{j=0}^{k} \binom{N}{j} \bmod (2MN + 2k + 1)$ using algorithm 2 below.*

   b *Compute $y = 5^{N-2}10^{n-N+2}x \bmod (2MN + 2k + 1)$.*

   c *Compute $c := \{c + (-1)^k y/(2MN + 2k + 1)\}$.*

4. *(Final step) Compute the value x as the fractional part of $b - c$ $(x = b - c - [b - c])$. Then x is an approximation of $\{10^n\pi\}$ with an error less than 10^{-n_0}.*

Notice that the floating point numbers involved should be encoded with a precision $10^{-n_0}/(2MN)$ to ensure the final required error bound.

Algorithm 1 requires the computation of sums of binomials modulo integers $m = 2MN+2k+1$. Binomials are iteratively calculated with the formula

$$\binom{N}{j} = \frac{n-j+1}{j}\binom{N}{j-1}.$$

The difficulty relies in the fact that the modulo number m can have prime factors smaller than j, thus inverting j modulo m is not always possible. We overcome this problem by taking the gcd of j with m at each step of the algorithm. The binomials will be decomposed in the form

$$\binom{N}{j} = \frac{A}{B} \times R_1 \times R_2 \times \cdots \times R_\ell$$

where A and B will not contain any prime factors $p \leq k$ of m, and each R_i is a power of the prime factor p_i of m. All this is detailed in the following algorithm.

Algorithm 2 (Sum of binomials modulo an integer) *Let m be a positive integer. This algorithm computes the value*

$$S = \sum_{j=0}^{k} \binom{N}{j} \bmod m.$$

1. *Compute the prime factors $p_1, \ldots, p_\ell$ of m (we restrict on prime factors p_i of m such that $p_i \leq k$).*
2. *Initialize $A = 1$, $B = 1$, $C = 1$, and $R_1 = \ldots = R_\ell = 1$.*
3. *For index j, $1 \leq j \leq k$, perform the following operations :*
   - a *Assign $a = n - j + 1$ and $b = j$.*
   - b *Decompose a and b with the powers of p_i, in the form*
   $$a = a^* \times p_1^{\alpha_1} \cdots p_\ell^{\alpha_\ell}, \qquad b = b^* \times p_1^{\beta_1} \cdots p_\ell^{\beta_\ell}.$$
   For each i, $p_i^{\alpha_i}$ is the exact power of p_i that divides a, $p_i^{\beta_i}$ is the exact power of p_i that divides b, so that a^* and b^* do not have one of the p_i as a prime factor.
   - c *For all i, $1 \leq i \leq \ell$, compute $R_i := R_i \times p_i^{\alpha_i}/p_i^{\beta_i}$ (R_i is necessarily an integer).*
   - d *Compute the values*
   $$A := A \times a^* \bmod m, \qquad B := B \times b^* \bmod m$$
   and
   $$C := C \times b^* + A \times R_1 \times \cdots \times R_\ell \bmod m.$$
4. *(Final step) Then the value of S modulo m is equal to $C/B \bmod m$ (here inversion of B modulo m is needed).*

The correctness of the algorithm easily follows from the fact that, after each step j of the main loop, we have

$$\frac{\binom{N}{j}}{R_1 \times \cdots \times R_\ell} \equiv \frac{A}{B} \bmod m \quad \text{and} \quad \sum_{i=0}^{j} \binom{N}{i} \equiv \frac{C}{B} \bmod m.$$

The property that the values R_i are always integers comes from the fact that R_i is the power of p_i that divides $\binom{N}{j}$, and the binomials have integer values. Inversion of B modulo m at the final step is possible since B never contains one of the prime factors p_i. Notice also that each R_i is smaller than N, since from a classical result of number theory, the power q of a prime number p in $\binom{N}{j}$ is equal to

$$q = \sum_{h>0, p^h \leq N} \left(\left[\frac{N}{p^h}\right] - \left[\frac{N-j}{p^h}\right] - \left[\frac{j}{p^h}\right] \right).$$

Since $0 \leq [N/p^h] - [(N-j)/p^h] - [j/p^h] \leq 1$, the value of q is at most equal to the maximal possible value of h, leading to $p^q \leq N$. Another on algorithm 2 is that it uses only one inversion modulo m, whose cost is $O(\log m)$ elementary operations. Finally, an easy optimization of algorithm 2 is obtained when $k > N/2$ using the identity $\sum_{j=0}^{k} \binom{N}{j} = 2^N - \sum_{j=0}^{N-k-1} \binom{N}{j}$.

3.2 Complexity of the algorithm

We now study carefully the complexity of algorithm 1 to prove theorem 1. We start by analyzing complexity of algorithm 2, which have the most significant contribution to the global cost.

Lemma 1 *For $k < N < m$, algorithm 2 has a memory need of $O(\log^2 m)$ bits and time complexity of*

$$Cost_2(m) = O(\log m) + O(k) + O(\lambda_k(m)\, k)$$

elementary operations on numbers of size $O(\log m)$, where $\lambda_k(m)$ is the number of distinct prime factors p of m such that $p \leq k$.

Proof : In the notations of algorithm 2, ℓ is equal to $\lambda_k(m)$. The algorithm uses $O(1+\ell)$ numbers of size $O(\log m)$; since $2^\ell \leq p_1 \times \cdots \times p_\ell \leq m$, we have $\ell = O(\log m)$ thus the memory usage is $O(\log^2 m)$.

As for the time complexity, we proceed as follows. For a given j, step 3b in algorithm 2 requires

$$\ell + \sum_i \alpha_i + \sum_i \beta_i$$

elementary operations of size $O(\log m)$. Considering a sequence of k consecutive integers, a given power p_i^h can be the factor of at most $1 + k/p_i^h$ of these numbers, thus for a given i the sum of the α_i and the β_i for all values of j is bounded by $O(k)$. Thus the total cost contribution of step 3b in algorithm 2 is $O(\ell k)$ elementary operations of size $O(\log m)$. For a given j, steps 3c and 3d have a cost of $O(1+\ell)$ and we conclude that the total cost of step 3 in algorithm 2 is $O(k + \ell k)$ elementary operations of size $O(\log m)$. Adding the cost $O(k)$ of step 1 and $O(\log n)$ of step 4, we obtain the result. •

We are now able to analyze the complexity of algorithm 1. First we concentrate on the cost of the computation of the term B defined by (9) in step 2 of the algorithm. The most representative cost is the powering in step 2a, and the associated complexity is $O(\log n)$ elementary operations modulo $2k+1$. Adding this cost $(M+1)N$ times gives a final cost of $C_2 = O(MN \log n)$ for step 2. As for the memory, it only involves a fixed number of numbers of size $O(\log n)$.

Step 3 of the algorithm should be studied more carefully. The main cost of each step k, $0 \leq k < N$, is the one of algorithm 2 used with $m = 2MN + 2k + 1$. The memory requirement is thus $O(\log^2 n)$. As for the time complexity, lemma 1 entails that it is clearly bounded by $O(N + N\lambda(m))$ where $\lambda(m)$ is the total number of distinct prime factors p of m with $p < N$. Thus the cost of step 3 in algorithm 1 is bounded by $O(N\Lambda)$, where

$$\Lambda = \sum_{k=0}^{N-1} \lambda(2MN + 2k + 1).$$

Any prime number $p \leq N$ can be the factor of at most $1 + N/p$ numbers in the arithmetic progression of the N integers $(2MN + 2k + 1)_{0 \leq k < N}$, thus

$$\Lambda \leq \sum_{p \text{ prime}, p<N} 1 + \frac{N}{p} \leq N + N \sum_{p \text{ prime}, p<2(M+1)N} \frac{1}{p}.$$

The inverse of primes smaller than x is known to be $O(\log\log x)$, thus we obtain $\Lambda = O(N \log\log N)$ and the total cost of step 3 is $O(N^2 \log\log n)$. Adding the cost of step 2, we have demonstrated that the cost of algorithm 1 is

$$\text{Cost}_1 = O(MN \log n) + O(N^2 \log\log n).$$

Since $M = O(n/\log^3 n)$ and $N = O(n/\log n)$, we have proved theorem t.1.

4 The n-th decimal digit computation with intermediate memory

On the one hand, we have been able to obtain an algorithm to compute directly the n-th decimal digit of π in nearly quadratic time and using only $O(\log^2 n)$ memory; on the other hand, computing all the first n digits of π is possible in quasi-linear time and with memory $O(n)$. It is natural to ask if it were possible to find intermediate algorithms, that use a limited amount of memory $O(m)$ (for example memory $O(\sqrt{n})$) and obtain an intermediate cost between linear and quadratic. In fact, the question has a positive answer by the use of formula (7) together with the so-called *binary splitting* technique on numbers of size $O(m)$. More precisely, we have the following result :

Theorem 2 *Let n_0 be a fixed (small) positive integer, $n \geq 4\,n_0$, let $\epsilon > 0$ be fixed, $\epsilon < 0.1$ and suppose we make use of an amount of $O(m)$ memory, with $n^\epsilon \leq m \leq n$. We can explicit an algorithm that computes the fractional part of $10^n\pi$ with a precision 10^{-n_0} with time complexity*

$$O\left(\frac{n^2 \log^3 n \log\log n}{m \log^2(n/m)}\right). \tag{11}$$

Here are some specializations of the results :

- When the memory available is $O(\sqrt{n})$, it is possible to compute the n-th decimal digit of π in time $O(n^{3/2} \log n \log\log n)$.
- The case where n and m are of the same order uses as much memory as used to compute all the n first digits of π, and the associated complexity is of the same order as the complexity of computing the first n digits π with binary splitting. However, we should expect that practical implementation should be significantly longer (a factor 10 is probably a first rough estimate of the lack of efficiency).

The details of the algorithm will be added soon in a next version of this unpublished paper.

5 Implementation and timings

An implementation of algorithm 1 has been made by the author with the program *pidec*. A computation of the 4,000,000-th decimal digit of π has been successfully obtained in less than two days on a Pentium III running at 900 Mhz. The corresponding source code and executable for windows is available on the author site [9]. Even if the complexity if sub-linear, the constant is front of the big O in theorem 1 is big and for reachable number of digits, the timings are worst than the one obtained with classical quadratic methods (arctan formula classic implementation). In figure 1, the timings are compared with the program by Fabrice Bellard [2], which runs in quadratic time.

A first look at this table shows that practical timings on *pidec* are consistent with the theoretical complexity of theorem 1. Another remark is that the quadratic contribution in Fabrice Bellard program running times is low. Nevertheless, the improvement of our technique is effective : for 5000 digits, pidec is 5 times faster, for 1 million digits it should be more than 10 times faster.

Index of decimal digit	*pidec* time	F. Bellard program time
5,000	0.96 sec	4.85 sec
10,000	3.13 sec	18.10 sec
20,000	10.34 sec	68.29 sec
40,000	35.96 sec	259.8 sec
100,000	185.1 sec	1520 sec
200,000	628.7 sec	5703 sec
500,000	3525 sec	34730 sec
1,000,000	15869 sec	not ran
2,000,000	42316 sec	not ran
4,000,000	168191 sec	not ran

Figure 1: Comparison of timings on the n-th decimal digit computation, between the pidec program by the author and Fabrice Bellard program, on a Pentium III 900 Mhz. (Fabrice Bellard program has been compiled using the `long long` option, which is faster).

6 Conclusion

The technique we have presented generalizes easily to other families of constants which are defined by alternating series, like

$$\begin{aligned}
\log(2) &= \sum_{n>0} \frac{(-1)^{n-1}}{n} \\
\frac{\pi^2}{16} &= \sum_{n>0} \frac{(-1)^{n-1}}{n^2} \\
\frac{7\pi^4}{720} &= \sum_{n>0} \frac{(-1)^{n-1}}{n^4} \\
\frac{3\zeta(3)}{4} &= \sum_{n>0} \frac{(-1)^{n-1}}{n^3} \\
\frac{\pi^3}{32} &= \sum_{n\geq 0} \frac{(-1)^{n}}{(2n+1)^3} \\
\cdots &\cdots \cdots
\end{aligned}$$

A large family of Bailey-Borwein-Plouffe like formulas also fit to our approach.

Another easy generalization is the computation of π in base B when B is even. The question of odd bases B is not solved by easy generalization and needs more investigations. A solution could be found by choosing another families of polynomials P_m ; instead of (5), one should choose a power of a polynomial for which the value at -1 is odd, but this is not enough to answer the problem.

Finally, the author thinks that in the followings years, distributed computations on home computers with algorithms like the one referenced in theorem 2 will be used to increase the reachable decimal digit of π with home computers, even if no quasi-linear complexity technique is found. Thousands of home computers could go higher than super computers ?

References

[1] D.H. Bailey, P.B. Borwein and S. Plouffe, *On the Rapid Computation of Various Polylogarithmic Constants*, Mathematics of Computation, (1997), vol. 66, p. 903-913

[2] F. Bellard, *Computation of the n'th digit of pi in any base in* $O(n^2)$, unpublished (1997) *http://fabrice.bellard.free.fr/pi/pi_n2/pi_n2.html*

[3] D. J. Broadhurst, *Polylogarithmic ladders, hypergeometric series and the ten millionth digits of* $\zeta(3)$ *and* $\zeta(5)$, preprint (1998).

[4] H. Cohen, F. Rodriguez Villegas, D. Zagier, *Convergence acceleration of alternating series*, preprint, Bonn, (1991)

[5] *Colin Percival PiHex project.* Home page at *http://www.cecm.sfu.ca/projects/pihex/pihex.html*

[6] Kanada Laboratory home page, at *ftp://pi.super-computing.org/*

[7] S. Plouffe, *On the computation of the n'th decimal digit of various transcendental numbers*, unpublished (November 1996) *http://www.lacim.uqam.ca/˜plouffe/Simon/articlepi.html*

[8] *PiFast, the fastest windows program to compute* π PiFast home page at *http://numbers.computation.free.fr/Constants/PiProgram/pifast.html*

[9] *N-th digit computation* In Xavier Gourdon and Pascal Sebah web site at *http://numbers.computation.free.fr/Constants/Algorithms/nthdigit.html*