MATHEMATICS OF COMPUTATION
VOLUME 50, NUMBER 181
JANUARY 1988, PAGES 283-296

The Computation of π to 29,360,000 Decimal Digits Using Borweins' Quartically Convergent Algorithm

By David H. Bailey

Abstract. In a recent work [6], Borwein and Borwein derived a class of algorithms based on the theory of elliptic integrals that yield very rapidly convergent approximations to elementary constants. The author has implemented Borweins' quartically convergent algorithm for $1/\pi$, using a prime modulus transform multi-precision technique, to compute over 29,360,000 digits of the decimal expansion of π. The result was checked by using a different algorithm, also due to the Borweins, that converges quadratically to π. These computations were performed as a system test of the Cray-2 operated by the Numerical Aerodynamical Simulation (NAS) Program at NASA Ames Research Center. The calculations were made possible by the very large memory of the Cray-2.

Until recently, the largest computation of the decimal expansion of π was due to Kanada and Tamura [12] of the University of Tokyo. In 1983 they computed approximately 16 million digits on a Hitachi S-810 computer. Late in 1985 Gosper [9] reported computing 17 million digits using a Symbolics workstation. Since the computation described in this paper was performed, Kanada has reported extending the computation of π to over 134 million digits (January 1987).

This paper describes the algorithms and techniques used in the author's computation, both for converging to π and for performing the required multi-precision arithmetic. The results of statistical analyses of the computed decimal expansion are also included.

1. Introduction. The computation of the numerical value of the constant π has been pursued for centuries for a variety of reasons, both practical and theoretical. Certainly, a value of π correct to 10 decimal places is sufficient for most "practical" applications. Occasionally, there is a need for double-precision or even multi-precision computations involving π and other elementary constants and functions in order to compensate for unusually severe numerical difficulties in an extended computation. However, the author is not aware of even a single case of a "practical" scientific computation that requires the value of π to more than about 100 decimal places.

Beyond immediate practicality, the decimal expansion of π has been of interest to mathematicians, who have still not been able to resolve the question of whether the digits in the expansion of π are "random". In particular, it is widely suspected that the decimal expansions of $\pi, e, \sqrt{2}, \sqrt{2\pi}$, and a host of related mathematical constants all have the property that the limiting frequency of any digit is one tenth, and that the limiting frequency of any n-long string of digits is 10^{-n}. Such a guaranteed property could, for instance, be the basis of a reliable pseudo-random number generator. Unfortunately, this assertion has not been proven in even one instance. Thus, there is a continuing interest in performing statistical analyses on

Received January 27, 1986; revised April 27, 1987.
1980 *Mathematics Subject Classification* (1985 *Revision*). Primary 11-04, 65-04.

the decimal expansions of these numbers to see if there is any irregularity that would suggest this assertion is false.

In recent years, the computation of the expansion of π has assumed the role as a standard test of computer integrity. If even one error occurs in the computation, then the result will almost certainly be completely in error after an initial correct section. On the other hand, if the result of the computation of π to even 100,000 decimal places is correct, then the computer has performed billions of operations without error. For this reason, programs that compute the decimal expansion of π are frequently used by both manufacturers and purchasers of new computer equipment to certify system reliability.

2. History. The first serious attempt to calculate an accurate value for the constant π was made by Archimedes, who approximated π by computing the areas of equilateral polygons with increasing numbers of sides. More recently, infinite series have been used. In 1671 Gregory discovered the arctangent series

$$\tan^{-1}(x) = x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + \cdots .$$

This discovery led to a number of rapidly convergent algorithms. In 1706 Machin used Gregory's series coupled with the identity

$$\pi = 16\tan^{-1}(1/5) - 4\tan^{-1}(1/239)$$

to compute 100 digits of π.

In the nearly 300 years since that time, most computations of the value of π, even those performed by computer, have employed some variation of this technique. For instance, a series based on the identity

$$\pi = 24\tan^{-1}(1/8) + 8\tan^{-1}(1/57) + 4\tan^{-1}(1/239)$$

was used in a computation of π to 100,000 decimal digits using an IBM 7090 in 1961 [15]. Readers interested in the history of the computation π are referred to Beckmann's entertaining book on the subject [2].

3. New Algorithms for π. Only very recently have algorithms been discovered that are fundamentally faster than the above techniques. In 1976 Brent [7] and Salamin [14] independently discovered an approximation algorithm based on elliptic integrals that yields quadratic convergence to π. With all of the previous techniques, the number of correct digits increases only linearly with the number of iterations performed. With this new algorithm, each additional iteration of the algorithm approximately *doubles* the number of correct digits. Kanada and Tamura employed this algorithm in 1983 to compute π to over 16 million decimal digits.

More recently, J. M. Borwein and P. B. Borwein [4] discovered another quadratically convergent algorithm for π, together with similar algorithms for fast computation of all the elementary functions. Their quadratically convergent algorithm for π can be stated as follows: Let $a_0 = \sqrt{2}$, $b_0 = 0$, $p_0 = 2 + \sqrt{2}$. Iterate

$$a_{k+1} = \frac{(\sqrt{a_k} + 1/\sqrt{a_k})}{2}, \quad b_{k+1} = \frac{\sqrt{a_k}(1 + b_k)}{a_k + b_k}, \quad p_{k+1} = \frac{p_k b_{k+1}(1 + a_{k+1})}{1 + b_{k+1}}.$$

Then p_k converges quadratically to π: Successive iterations of this algorithm yield 3, 8, 19, 40, 83, 170, 345, 694, 1392, and 2788 correct digits of the expansion of π.

However, it should be noted that this algorithm is not self-correcting for numerical errors, so that all iterations must be performed to full precision. In other words, in a computation of π to 2788 decimal digits using the above algorithm, each of the ten iterations must be performed with more than 2788 digits of precision.

Most recently, the Borweins [6] have discovered a general technique for obtaining even higher-order convergent algorithms for certain elementary constants. Their quartically convergent algorithm for $1/\pi$ can be stated as follows: Let $a_0 = 6-4\sqrt{2}$ and $y_0 = \sqrt{2} - 1$. Iterate

$$\begin{aligned} y_{k+1} &= \frac{1-(1-y_k^4)^{1/4}}{1+(1-y_k^4)^{1/4}}, \\ a_{k+1} &= a_k(1+y_{k+1})^4 - 2^{2k+3}y_{k+1}(1+y_{k+1}+y_{k+1}^2). \end{aligned}$$

Then a_k converges quartically to $1/\pi$: Each successive iteration approximately *quadruples* the number of correct digits in the result. As in the previous case, each iteration must be performed to at least the level of precision desired for the final result.

4. Multi-Precision Arithmetic Techniques. A key element of a very high precision computation of this sort is a set of high-performance routines for performing multi-precision arithmetic. A naive approach to multi-precision computation would require a prohibitive amount of processing time and would, as a result, sharply increase the probability that an undetected hardware error would occur, rendering the result invalid. In addition to employing advanced algorithms for such key operations as multi-precision multiplication, it is imperative that these algorithms be implemented in a style that is conducive for high-speed computation on the computer being used.

The computer used for these computations is the Cray-2 at the NASA Ames Research Center. This computation was performed to test the integrity of the Cray-2 hardware, as well as the Fortran compiler and the operating system. The Cray-2 is particularly well suited for this computation because of its very large main memory, which holds $2^{28} = 268,435,456$ words (one word is 64 bits of data). With this huge capacity, all data for these computations can be contained entirely within main memory, insuring ease of programming and fast execution.

No attempt was made to employ more than one of the four central processing units in the Cray-2. Thus, at the same time these calculations were being performed, the computer was executing other jobs on the other processors. However, full advantage was taken of the vector operations and vector registers of the system. Considerable care was taken in programming to insure that the multi-precision algorithms were implemented in a style that would admit vector processing. Most key loops were automatically vectorized by the Cray-2 Fortran compiler. For those few that were not automatically vectorized, compiler directives were inserted to force vectorization. As a result of this effort, virtually all arithmetic operations were performed in vector mode, which on the Cray-2 is approximately 20 times faster than scalar mode. Because of the high level of vectorization that was achieved using the Fortran compiler, it was not necessary to use assembly language, nonstandard constructs, or library subroutines.

A multi-precision number is represented in these computations as an $(n+2)$-long array of floating-point whole numbers. The first cell contains the sign of the number, either 1, -1, or 0 (reserved for an exact zero). The second cell of the array contains the exponent (powers of the radix), and the remaining n cells contain the mantissa. The radix selected for the multi-precision numbers is 10^7. Thus the number 1.23456789 is represented by the array $1, 0, 1, 2345678, 9000000, 0, 0, \ldots, 0$.

A floating-point representation was chosen instead of an integer representation because the hardware of numerical supercomputers such as the Cray-2 is designed for floating-point computation. Indeed, the Cray-2 does not even have full-word integer multiply or divide hardware instructions. Such operations are performed by first converting the operands to floating-point form, using the floating-point unit, and converting the results back to fixed-point (integer) form. A decimal radix was chosen instead of a binary value because multiplications and divisions by powers of two are not performed any faster than normal on the Cary-2 (in vector mode). Since a decimal radix is clearly preferable to a binary radix for program troubleshooting and for input and output, a decimal radix was chosen. The value 10^7 was chosen because it is the largest power of ten that will fit in half of the mantissa of a single word. In this way two of these numbers may be multiplied to obtain the exact product using ordinary single-precision arithmetic.

Multi-precision addition and subtraction are not computationally expensive compared to multiplication, division, and square root extraction. Thus, simple algorithms suffice to perform addition and subtraction. The only part of these operations that is not immediately conducive to vector processing is releasing the carries for the final result. This is because the normal "schoolboy" approach of beginning at the last cell and working forward is a recursive operation. On a vector supercomputer this is better done by starting at the beginning and releasing the carry only one cell back for each cell processed. Unfortunately, it cannot be guaranteed that one application of this process will release all carries (consider the case of two or more consecutive 9999999's, followed by a number exceeding 10^7). Thus it is necessary to repeat this operation until all carries have been released (usually one or two additional times). In the rare cases where three applications of this vectorized process are not successful in releasing all carries, the author's program resorts to the scalar "schoolboy" method.

Provided a fast multi-precision multiplication procedure is available, multi-precision division and square root extraction may be performed economically using Newton's iteration, as follows. Let x_0 and y_0 be initial approximations to the reciprocal of a and to the reciprocal of the square root of a, respectively. Then

$$x_{k+1} = x_k(2 - a x_k), \qquad y_{k+1} = \frac{y_k(3 - a y_k^2)}{2}$$

both converge quadratically to the desired values. One additional full-precision multiplication yields the quotient and the square root, respectively. What is especially attractive about these algorithms is that the first iteration may be performed using ordinary single-precision arithmetic, and subsequent iterations may be performed using a level of precision that approximately doubles each time. Thus the total cost of computation is only about twice the cost of the final iteration, plus the one additional multiplication. As a result, a multi-precision division costs only about five times as much as a multi-precision multiplication, and a multi-precision square root costs only about seven times as much as a multi-precision multiplication.

5. Multi-Precision Multiplication. It can be seen from the above that the key component of a high-performance multi-precision arithmetic system is the multiply operation. For modest levels of precision (fewer than about 1000 digits), some variation of the usual "schoolboy" method is sufficient, although care must be taken in the implementation to insure that the operations are vectorizable. Above this level of precision, however, other more sophisticated techniques have a significant advantage. The history of the development of high-performance multiply algorithms will not be reviewed here. The interested reader is referred to Knuth [13]. It will suffice to note that all of the current state-of-the-art techniques derive from the following fact of Fourier analysis: Let $F(x)$ denote the discrete Fourier transform of the sequence $x = (x_0, x_1, x_2, \ldots, x_{N-1})$, and let $F^{-1}(x)$ denote the inverse discrete Fourier transform of x:

$$F_k(x) = \sum_{j=0}^{N-1} x_j \omega^{jk}, \qquad F_k^{-1}(x) = \frac{1}{N} \sum_{j=0}^{N-1} x_j \omega^{-jk},$$

where $\omega = e^{-2\pi i/N}$ is a primitive Nth root of unity. Let $C(x, y)$ denote the convolution of the sequences x and y:

$$C_k(x, y) = \sum_{j=0}^{N-1} x_j y_{k-j},$$

where the subscript $k-j$ is to be interpreted as $k-j+N$ if $k-j$ is negative. Then the "convolution theorem", whose proof is a straightforward exercise, states that

$$F[C(x, y)] = F(x)F(y),$$

or expressed another way,

$$C(x, y) = F^{-1}[F(x)F(y)].$$

This result is applicable to multi-precision multiplication in the following way. Let x and y be n-long representations of two multi-precision numbers (without the sign or exponent words). Extend x and y to length $2n$ by appending n zeros at the end of each. Then the multi-precision product z of x and y, except for releasing the carries, can be written as follows:

$$\begin{aligned}
z_0 &= x_0 y_0 \\
z_1 &= x_0 y_1 + x_1 y_0 \\
z_2 &= x_0 y_2 + x_1 y_1 + x_2 y_0 \\
&\vdots \\
z_{n-1} &= x_0 y_{n-1} + x_1 y_{n-2} + \cdots + x_{n-1} y_0 \\
&\vdots \\
z_{2n-3} &= x_{n-1} y_{n-2} + x_{n-2} y_{n-1} \\
z_{2n-2} &= x_{n-1} y_{n-1} \\
z_{2n-1} &= 0.
\end{aligned}$$

It can now be seen that this "multiplication pyramid" is precisely the convolution of the two sequences x and y, where $N = 2n$. The convolution theorem states that the multiplication pyramid can be obtained by performing two forward discrete Fourier transforms, one vector complex multiplication, and one reverse transform, each of length $N = 2n$. Once the resulting complex numbers have been rounded to the nearest integer, the final multi-precision product may be obtained by merely releasing the carries as described in the section above on addition and subtraction.

The key computational savings here is that the discrete Fourier transform may of course be economically computed using some variation of the "fast Fourier transform" (FFT) algorithm. It is most convenient to employ the radix two fast Fourier transform since there is a wealth of literature on how to efficiently implement this algorithm (see [1], [8], and [16]). Thus, it will be assumed from this point that $N = 2^m$ for some integer m.

One useful "trick" can be employed to further reduce the computational requirement for complex transforms. Note that the input data vectors x and y and the result vector z are purely real. This fact can be exploited by using a simple procedure ([8, p. 169]) for performing real-to-complex and complex-to-real transforms that obtains the result with only about half the work otherwise required.

One important item has been omitted from the above discussion. If the radix 10^7 is used, then the product of two cells will be in the neighborhood of 10^{14}, and the sum of a large number of these products cannot be represented exactly in the 48-bit mantissa of a Cray-2 floating-point word. In this case the rounding operation at the completion of the transform will not be able to recover the exact whole number result. As a result, for the complex transform method to work correctly, it is necessary to alter the above scheme slightly. The simplest solution is to use the radix 10^6 and to divide all input data into two words with only three digits each. Although this scheme greatly increases the memory space required, it does permit the complex transform method to be used for multi-precision computation up to several million digits on the Cray-2.

6. Prime Modulus Transforms. Some variation of the above method has been used in almost all high-performance multi-precision computer programs, including the program used by Kanada and Tamura. However, it appears to break down for very high-precision computation (beyond about ten million digits on the Cray-2), due to the round-off error problem mentioned above. The input data can be further divided into two digits per word or even one digit per word, but only with a substantial increase in run time and main memory. Since a principal goal in this computation was to remain totally within the Cray-2 main memory, a somewhat different method was used.

It can readily be seen that the technique of the previous section, including the usage of a fast Fourier transform algorithm, can be applied in any number field in which there exists a primitive Nth root of unity ω. This requirement holds for the field of the integers modulo p, where p is a prime of the form $p = kN + 1$ ([11, p. 85]). One significant advantage of using a prime modulus field instead of the field of complex numbers is that there is no need to worry about round-off error in the results, since all computations are exact.

However, there are some difficulties in using a prime modulus field for the transform operations above. The first is to find a prime p of the form $kN + 1$, where $N = 2^m$. The second is to find a primitive Nth root of unity modulo p. As it turns out, it is not too hard using a computer to find both of these numbers by direct search. Thirdly, one must compute the multiplicative inverse of N modulo p. This can be done using a variation of the Euclidean algorithm from elementary number theory. Note that each of these calculations needs to be performed one time only.

A more troublesome difficulty in using a prime modulus transform is the fact that the final multiplication pyramid results are only recovered modulo p. If p is greater than about 10^{24} then this is not a problem, but the usage of such a large prime would require *quadruple*-precision arithmetic operations to be performed in the inner loop of the fast Fourier transform, which would very greatly increase the run time. A simpler and faster approach to the problem is to use two primes, p_1 and p_2, each slightly greater than 10^{12}, and to perform the transform algorithm above using each prime. Then the Chinese remainder theorem may be applied to the results modulo p_1 and p_2 to obtain the results modulo the product p_1p_2. Since p_1p_2 is greater than 10^{24}, these results will be the exact multiplication pyramid numbers. Unfortunately, double-precision arithmetic must still be performed in the fast Fourier transform and in the Chinese remainder theorem calculation. However, the whole-number format of the input data simplifies these operations, and it is possible to program them in a vectorizable fashion.

Borodin and Munro ([3, p. 90]) have suggested using three transforms with three primes p_1, p_2 and p_3, each of which is just smaller than half of the mantissa, and using the Chinese remainder theorem to recover the results modulo $p_1p_2p_3$. In this way, double-precision operations are completely avoided in the inner loop of the FFT. This scheme runs quite fast, but unfortunately the largest transform that can be performed on the Cray-2 using this system is $N = 2^{19}$, which corresponds to a maximum precision of about three million digits.

Readers interested in studying about prime modulus number fields, the Euclidean algorithm, or the Chinese remainder theorem are referred to any elementary text on number theory, such as [10] or [11]. Knuth [13] and Borodin [3] also provide excellent information on using these tools for computation.

7. Computational Results. The author has implemented all three of the above techniques for multi-precision multiplication on the Cray-2. By employing special high-performance techniques [1], the complex transform can be made to run the fastest, about four times faster than the two-prime transform method. However, the memory requirement of the two-prime scheme is significantly less than either the three-prime or the complex scheme, and since the two-prime scheme permits very high-precision computation, it was selected for the computations of π.

One of the author's computations used twelve iterations of Borweins' quartic algorithm for $1/\pi$, followed by a reciprocal operation, to yield 29,360,128 digits of π. In this computation, approximately 12 trillion arithmetic operations were performed. The run took 28 hours of processing time on one of the four Cray-2 central processing units and used 138 million words of main memory. It was started on January 7, 1986 and completed January 9, 1986. The program was not running this entire time—the system was taken down for service several times, and the run was frequently interrupted by other programs. Restarting the computation after a system down was a simple matter since the two key multi-precision number arrays were saved on disk after the completion of each iteration.

This computation was checked using 24 iterations of Borweins' quadratically convergent algorithm for π. This run took 40 hours processing time and 147 million words of main memory. A comparison of these output results with the first run found no discrepancies except for the last 24 digits, a normal truncation error. Thus it can be safely assumed that at least 29,360,000 digits of the final result are correct.

It was discovered after both computations were completed that one loop in the Chinese remainder theorem computation was inadvertently performed in scalar mode instead of vector mode. As a result, both of these calculations used about 25% more run time than would otherwise have been required. This error, however, did not affect the validity of the computed decimal expansions.

8. Statistical Analysis of π. Probably the most significant mathematical motivation for the computation of π, both historically and in modern times, has been to investigate the question of the randomness of its decimal expansion. Before Lambert proved in 1766 that π is irrational, there was great interest in checking whether or not its decimal expansion eventually repeats, thus disclosing that π is rational. Since that time there has been a continuing interest in the still unanswered question of whether the expansion is statistically random. It is of course strongly suspected that the decimal expansion of π, if computed to sufficiently high precision, will pass any reasonable statistical test for randomness. The most frequently mentioned conjecture along this line is that any sequence of n digits occurs with a limiting frequency of 10^{-n}.

With 29,360,000 digits, the frequencies of n-long strings may be studied for randomness for n as high as six. Beyond that level the expected number of any one string is too low for statistical tests to be meaningful. The results of tabulated frequencies for one and two digit strings are listed in Tables 1 and 2. In the first table the Z-score numbers are computed as the deviation from the mean divided by the standard deviation, and thus these statistics should be normally distributed with mean zero and variance one.

TABLE 1

Single digit statistics

Digit	Count	Deviation	Z-score
0	2935072	− 928	− 0.5709
1	2936516	516	0.3174
2	2936843	843	0.5186
3	2935205	− 795	− 0.4891
4	2938787	2787	1.7145
5	2936197	197	0.1212
6	2935504	− 496	− 0.3051
7	2934083	− 1917	− 1.1793
8	2935698	− 302	− 0.1858
9	2936095	95	0.0584

TABLE 2

Two digit frequency counts

00	293062	01	293970	02	293533	03	292893	04	294459
05	294189	06	292688	07	292707	08	294260	09	293311
10	294503	11	293409	12	293591	13	294285	14	294020
15	293158	16	293799	17	293020	18	293262	19	293469
20	293952	21	293226	22	293844	23	293382	24	293869
25	293721	26	293655	27	293969	28	293320	29	293905
30	293718	31	293542	32	293272	33	293422	34	293178
35	293490	36	293484	37	292694	38	294152	39	294253
40	294622	41	294793	42	293863	43	293041	44	293519
45	293998	46	294418	47	293616	48	293296	49	293621
50	292736	51	294272	52	293614	53	293215	54	293569
55	294194	56	293260	57	294152	58	293137	59	294048
60	293842	61	293105	62	294187	63	293809	64	293463
65	293544	66	293123	67	293307	68	293602	69	293522
70	292650	71	294304	72	293497	73	293761	74	293960
75	293199	76	293597	77	292745	78	293223	79	293147
80	292517	81	292986	82	293637	83	294475	84	294267
85	293600	86	293786	87	293971	88	293434	89	293025
90	293470	91	292908	92	293806	93	292922	94	294483
95	293104	96	293694	97	293902	98	294012	99	293794

The most appropriate statistical procedure for testing the hypothesis that the empirical frequencies of n-long strings of digits are random is the χ^2 test. The χ^2 statistic of the k observations $X_1, X_2, \ldots, X_k$ is defined as

$$\chi^2 = \sum_{i=1}^{k} \frac{(X_i - E_i)^2}{E_i}$$

where E_i is the expected value of the random variable X_i. In this case $k = 10^n$ and $E_i = 10^{-n} d$ for all i, where $d = 29,360,000$ denotes the number of digits. The mean of the χ^2 statistic in this case is $k - 1$ and its standard deviation is $\sqrt{2(k-1)}$. Its distribution is nearly normal for large k. The results of the χ^2 analysis are shown in Table 3.

TABLE 3

Multiple digit χ^2 statistics

Length	χ^2 value	Z-score
1	4.869696	– 0.9735
2	84.52604	– 1.0286
3	983.9108	– 0.3376
4	10147.258	1.0484
5	100257.92	0.5790
6	1000827.7	0.5860

Another test that is frequently performed on long pseudo-random sequences is an analysis to check whether the number of n-long repeats for various n is within statistical bounds of randomness. An n-long repeat is said to occur if the n-long digit sequence beginning at two different positions is the same. The mean M and the variance V of the number of n-long repeats in d digits are (to an excellent approximation)

$$M = \frac{10^{-n}d^2}{2}, \qquad V = \frac{11 \cdot 10^{-n}d^2}{18}.$$

Tabulation of repeats in the expansion of π was performed by packing the string beginning at each position into a single Cray-2 word, sorting the resulting array, and counting equal contiguous entries in the sorted list. The results of this analysis are shown in Table 4.

TABLE 4

Long repeat statistics

10	42945	43100.	– 0.677
11	4385	4310.	1.033
12	447	431.	0.697
13	48	43.1	0.675
14	6	4.31	0.736
15	1	0.43	0.784

A third test frequently performed as a test for randomness is the runs test. This test compares the observed frequency of long runs of a single digit with the number of such occurrences that would be expected at random. The mean and variance of this statistic are the same as the formulas for repeats, except that d^2 is replaced by $2d$. Table 5 lists the observed frequencies of runs for the calculated expansion of π.

The frequencies of long runs are all within acceptable limits of randomness. The only phenomenon of any note in Table 5 is the occurrence of a 9-long run of sevens. However, there is a 29% chance that a 9-long run of some digit would occur in 29,360,000 digits, so this instance by itself is not remarkable.

TABLE 5

Single-digit run counts

	Length of Run				
Digit	5	6	7	8	9
0	308	29	3	0	0
1	281	21	1	0	0
2	272	23	0	0	0
3	266	26	5	0	0
4	296	40	6	1	0
5	292	30	4	0	0
6	316	33	3	0	0
7	315	37	6	2	1
8	295	36	3	0	0
9	306	40	7	0	0

9. Conclusion. The statistical analyses that have been performed on the expansion of π to 29,360,000 decimal places have not disclosed any irregularity. The observed frequencies of n-long strings of digits for n up to 6 are entirely unremarkable. The numbers of long repeating strings and single-digit r··ns are completely acceptable. Thus, based on these tests, the decimal expansion of π appears to be completely random.

Appendix

Selected Output Listing

```
Initial 1000 digits:

3.
14159265358979323846264338327950288419716939937510
58209749445923078164062862089986280348253421170679
82148086513282306647093844609550582231725359408128
48111745028410270193852110555964462294895493038196
44288109756659334461284756482337867831652712019091
45648566923460348610454326648213393607260249141273
72458700660631558817488152092096282925409171536436
78925903600113305305488204665213841469519415116094
33057270365759591953092186117381932611793105118548
07446237996274956735188575272489122793818301194912
98336733624406566430860213949463952247371907021798
60943702770539217176293176752384674818467669405132
00056812714526356082778577134275778960917363717872
14684409012249534301465495853710507922796892589235
42019956112129021960864034418159813629774771309960
51870721134999999837297804995105973173281609631859
50244594553469083026425223082533446850352619311881
71010003137838752886587533208381420617177669147303
59825349042875546873115956286388235378759375195778
18577805321712268066130019278766111959092164201989

Digits 4,999,001 to 5,000,000:

49480754784558100182731931632488412804488722296956
79855015464855780486736535227902836997918084867230
64962221004527085768335035212069684801817137616329
97561738425160340472537100056351640342162492027179
66824926458930960182645026923102266570541641475347
20341554913770421505764452807809035248393621093031
02288096238486877923145240841637271180953058890040
68843766781431498914299893621278545260143140439048
49938801556336059513116731891132765777881364690708
47036863411196323063886507480852125682842257852524
03086993703255692093960818587414181230484153204049
20234989002732447593020323794790776444752398445514
67304403210968985244961967143433964895893190552338
49818852746844924836314634250006421630628686858848
27453318669926734730642735036364002856022218966350
11429182634319974163253372368798553451111253055262
39104082639970934508146672521381105913047210052428
18988626533169469331951675296209306752291590715999
89846179288059262000848638138811280944056488021060
48865855191846702365421761783505181721320764619715
```

Digits 9,999,001 to 10,000,000:

5509781824351672822784991072040028675790790446633 5
1271820297952515061772533406689498895642470326923 0

1539982090039016627522433818442480858939529365258 2

5363565858417548553674481865028924518820644785328 0

7912967550486557292908308348548393758333467101908 9

1206711453695517314092946182346647872528952997420 4

0212763523592329330577017942386522596324069402748 0

6041288030309245248103494158273593244388727310939 7

4163488960469581924539515134104343399838187465097 2

3369263522579147245424440132631296439639120960780 0

1634485119912542081973740744604589974214573104231 3

6445648650193780106352660374405656882386138937544 3

9735168129683156791161888422225114147732261233139 6

1860608037311034869266093394043841630032614344928 0

5082113157573772773982155152228650999766243258721 3

9339344590209166227290549349382717820512666902114 9

4719231138093382231122409958837224633250122232337 8

9689526902536626394126701031732786498717025714961 7

7610515549257985759204553246894468742702504639790 5

6532655319406099946978733381063171948173534895589 7

Digits 14,999,001 to 15,000,000:

7516191258272903443712327974925631151192524395698 5

4146673506919481516383722607392515188775175165974 1

0062288072644860220945693041448853988298110851249 2

3062608837596678362164975341253968308492271134251 3

9495399569362544133140173813308584817231588747322 5

6686213925193854010224947557549494715839562351278 5

6703388882449555108446230047240761216595278438625 2

8305999230222328486593456626292974843682773081203 0

1443459368987425976641551441209798413399801593458 4

3539347565062432385016043273191880512640667187135 3

7755576621467093181315116287950050971055179515281 8

0909315448105804476736412216610003242509826316625 7

4173051822048071548822461656389134404693420810323 8

3990325402988174634249658318683694748619425753354 0

3633122383822239249405627085637803305621354468659 3

0298682171495280858594941867653229106733981768485 0

7757615178505727709988062737081438579411766876359 9

7581449914989031459409852596033637798998822813857 9

0395460850007618075488043395846861964109276265344 6

7964520526347339328607497932393150314117277566980 3

Digits 19,999,001 to 20,000,000:

24662421652199659486815804456870197576438951607697
86758526528445124126249995515004465281646092893016
37396198596248627116552469686381679679898926165214
19985145392716546108714664257998278750239431446690
24524827883001435830699295155565194378002452231513
03498450165135282534109758167508041457187906821950
98156889669401540575560430489547131781464796920586
99611799897126388736531564345333853581593559913668
62608486227029865668230856391322081859205243349223
41898466479821052634622968628766495150696262416056
24275201300452308788083860012754008114751496913646
62422297630443481605116791864334302662386921297850
27885235888942133721123400642720173755448172632485
38990548569368292370090889371435442648824207842546
28067400727949203553263884395310176843535902614634
76307233029969045465206192626213143248919480318684
24091340888618503237670440877047193079665717842568
49026897445701681738816789861189706430445720674936
81903857815020793466156644931359073005891342758785
95072447895232808191116291055801380049338634527644

Digits 24,999,001 to 25,000,000:

64626376657788401626872035835150250932381126804132
24527774629670113871130617683224437149346115597163
91099108362268853888484703799982396604187954247350
36635859521304516872709809678948655853409228442863
24948936001342207955968740967092110719683856558205
30816048151902240856062148774123551023529985810792
74189214723685203602121713995138514107079374902532
54350785997288413483911434952219864948321330490074
60146435121254311259573947301142531184570914224080
72612210306331872567179327168155609249989038137333
66960257521334843154895361888436208731274888674781
18373984739313750077149269011462219615798047067514
35050981335283641909759090614464729227662129370246
47057090874450108027231969863517024941726518038367
32762891741863822149208539226376382907305941739639
07549588865849168186491743776278287261919660505923
92475738836587226649359524383297861404378228288281
73596312642574370611956801297356036342637793562761
38037507909491563108238168922672241753290045253446
07864115924597806944245511285225546774836191884322

```
Digits 29,359,001 to 29,360,000:

3419284178891522964336847388197769853900574621984б
69525347577001729886543392436261840972591968259157
6110747629400730307400523562782978702554407540554З
9989507153059816218961131505041969730972829060606Z
1889011613820684258998021544539575359379289882357S
0141234748667204693563573577738064843730857329184O
6210849633097482768941126867522297552323062395683З
6263114891606388397766197309149915519284789410969l
396122653293511959787255667642564628953751809074 49
4936309292131412764088851017042258408474414931911 8
6575582572177283614497797876605228546904719759626 4
7668005536084220968951773713500861189045243301521 2
3769374570207033898894012337669396105726953527814 6
9971913630707464320185386407130799750797450988355 4
6596157578284974751264578644113084532532314940541 9
1726336489964791203287817189338731781932491238234 2
1864827176372302256172001634836858495565816511248 9
9544684872069362195779794342949464025841993908913 5
3426698523277623931436525967083202637025092477681 4
7049097142449367541433098725950780665432227288825 3
```

NAS Systems Division
NASA Ames Research Center
Mail Stop 258-5
Moffett Field, California 94035

1. D. H. BAILEY, "A high-performance fast Fourier transform algorithm for the Cray-2," *J. Supercomputing*, v. 1, 1987, pp. 43–60.

2. P. BECKMANN, *A History of Pi*, Golem Press, Boulder, CO, 1971.

3. A. BORODIN & I. MUNRO, *The Computational Complexity of Algebraic and Numeric Problems*, American Elsevier, New York, 1975.

4. J. M. BORWEIN & P. B. BORWEIN, "The arithmetic-geometric mean and fast computation of elementary functions," *SIAM Rev.*, v. 26, 1984, pp. 351–366.

5. J. M. BORWEIN & P. B. BORWEIN, "More quadratically converging algorithms for π," *Math. Comp.*, v. 46, 1986, pp. 247–253.

6. J. M. BORWEIN & P. B. BORWEIN, *Pi and the AGM—A Study in Analytic Number Theory and Computational Complexity*, Wiley, New York, 1987.

7. R. P. BRENT, "Fast multiple-precision evaluation of elementary functions," *J. Assoc. Comput. Mach.*, v. 23, 1976, pp. 242–251.

8. E. O. BRIGHAM, *The Fast Fourier Transform*, Prentice-Hall, Englewood Cliffs, N. J., 1974.

9. W. GOSPER, private communication.

10. EMIL GROSSWALD, *Topics from the Theory of Numbers*, Macmillan, New York, 1966.

11. G. H. HARDY & E. M. WRIGHT, *An Introduction to the Theory of Numbers*, 5th ed., Oxford Univ. Press, London, 1984.

12. Y. KANADA & Y. TAMURA, *Calculation of* π *to* 10,013,395 *Decimal Places Based on the Gauss-Legendre Algorithm and Gauss Arctangent Relation*, Computer Centre, University of Tokyo, 1983.

13. D. KNUTH, *The Art of Computer Programming*, Vol. 2: *Seminumerical Algorithms*, Addison-Wesley, Reading, Mass., 1981.

14. E. SALAMIN, "Computation of π using arithmetic-geometric mean," *Math. Comp.*, v. 30, 1976, pp. 565–570.

15. D. SHANKS & J. W. WRENCH, JR., "Calculation of π to 100,000 decimals," *Math. Comp.*, v. 16, 1962, pp. 76–99.

16. P. SWARZTRAUBER, "FFT algorithms for vector computers," *Parallel Comput.*, v. 1, 1984, pp. 45–64.