Memory Efficient Arithmetic

Ernie Croot

February 3, 2007

Abstract

In this paper we give an algorithm for finding the mth base-b digit of a positive integer n ($m = 1$ is the least significant digit) defined as the final number in a sequence of integers gotten by multiplying, adding, and subtracting previous numbers in the sequence (actually, the algorithm finds arbitrarily precise approximations to $n/b^m \pmod 1$, which can be used to get this mth digit whenever the lower $m-1$ digits do not begin with a long run of the digit $b-1$). In many cases, this algorithm will require far less memory than it takes to write down the base-b digits of n, and will have a running time (in bit operations) only slightly worse than linear in the number of these base-b digits.

One easy-to-state consequence of the above result is that the mth base-10 digit of 2^t can be found using $O(t^{2/3}\log^C t)$ bits of memory and $O(t\log^C t)$ bit operations, where $C > 0$ is constant. Of course, if $m = O(t^{2/3})$ then one can do much better (by just computing $n \pmod{b^m}$), so the result is only non-trivial when $t = O(m^{3/2})$.

The algorithm we give is highly parallelizable, although if one uses M processors, to get an M-fold reduction in running time, the memory requirements will increase by a factor of M.

1 Introduction

Suppose that α is a positive real number. Then, a standard fact regarding base-b representations is that given any integer $b \geq 2$, there exists and integer J, and a sequence of integers $r_J, r_{J-1}, ...$, such that

$$\alpha \ = \ \sum_{j \leq J} r_j b^j, \ 0 \leq r_j \leq b-1, \ r_J \neq 0, \tag{1}$$

and we write

$$\alpha \;=\; (r_J r_{J-1} ... r_0 . r_{-1} ...)_b$$

to denote this expansion (when $J > 0$). If we further disallow $r_n = b - 1$ for all $n \leq N$, for some N, then the sequence of r_i's is unique. If we make no such restrictions on r_n, then the sequence of r_i's may not be unique, as in the base-10 expansion

$$1 \;=\; 0.9999999...$$

A question which has attracted recent attention (see [1]) is the following: Given a base b, and an integer $m \geq 0$, efficiently determine a good approximation to

$$\nu \;=\; \{b^m \pi\},$$

where for a real number θ, the notation $\{\theta\}$ means

$$\{\theta\} \;=\; \theta \;-\; \lfloor \theta \rfloor \quad (\text{Note}: \;\; \{\theta\} \equiv \theta \pmod 1, \;\; 0 \leq \{\theta\} < 1).$$

We will say that such an approximation γ is a *level-p approximation* if and only if

$$|\gamma - \nu| \;<\; \frac{1}{b^p}.$$

Now, if the approximation γ is sufficiently good, then one can use it to determine the digits of π: For example, if we take $m = 3$ and $b = 10$, then we get

$$\nu \;\equiv\; b^m \pi \;\equiv\; 10^3 \pi \;\equiv\; 0.59265358979... \pmod 1.$$

Suppose we had an approximation γ to ν satisfying

$$|\gamma \;-\; \nu| \;<\; \frac{1}{1000};$$

so, $\gamma = 0.59....$ Then, the leading base-10 digit of γ is the same as the fifth base-10 digit of π (from the left), which is 5.

More generally, a good approximation to ν gives us the $(m+2)$nd digit of π; however, depending on the value of m selected, this approximation may need to be extremely close to ν, in order to determine this digit. For example, in the above instance with $m = 3$, if $\gamma = 0.6$, then

$$|\gamma \;-\; \nu| \;<\; \frac{1}{100},$$

and we note that the leading digit of γ is not the same as the leading digit of ν.

In this paper we will describe a method for determining the mth base-b digit of an integer, where $m = 1$ corresponds to the least significant digit (note that for the digits-of-π problem above, $m = 1$ corresponded to the leading digit; so, the mth digit is defined differently in this context). As in the problem concerning digits of π, this method produces arbitrarily precise approximations to

$$\nu \;=\; \left\{ \frac{n}{b^m} \right\},$$

where n is this integer, which will be defined by a certain type of expansion that we will describe below. Note that the value of m is at least 1 (if $m \leq 0$, then ν is trivially 0). Now, if

$$\nu \;=\; (0.r_{-1}r_{-2}\cdots)_b,$$

then the mth digit of n equals r_{-1}; and, if λ is a sufficiently good approximation to ν, then the leading digit of λ will also equal r_{-1}.

The type of expansion for n we will use is defined as follows: In [2], Smale and Shub say that a *computation of length* L of a positive integer n is a sequence of integers $s_1, s_2, ..., s_L$, where $s_1 = 0$, $s_2 = 1$, and for $i \geq 3$,

$$s_i \;=\; s_j \circ s_k, \text{ where } j, k < i,$$

where $\circ$ is either addition, subtraction, or multiplication, and where $s_L = n$. If one expresses such a computation as a string indices (j, k) and operations $+, -$ and $\times$, then given an integer $N \geq 2$, one can compute $n \pmod N$ using only $O(L(\log L)(\log N)(\log\log N)^2)$ bit operations, by just computing the sequence $s_1, ..., s_L$ modulo N. The factor $(\log N)(\log\log N)^2$ in this big-O appears because a product of integers modulo N can be computed using Fast Fourier Transforms using only $O((\log N)(\log\log N)^2)$ bit operations.

We will say that an integer n has *computational complexity* L if and only if there exists a computation of lenth L for computing n.

In the next section we will prove a general result, which we will use to prove the following theorem:

Theorem 1 *Given a positive integer n having computational complexity L, an approximation A to the number of base-b digits of n (see the input specs*

below), an integer $m \geq 0$, and a level y, there exists an algorithm for computing a level-y approximation to

$$\nu \;=\; \left\{\frac{n}{b^m}\right\}.$$

This algorithm requires only

$$O(yL(\log^{2/3} n)\log^C(b+m+y+L+\log n)) \text{ bits of memory,}$$

and

$$O(yL(\log n)\log^C(b+m+y+L+\log n)) \text{ bit operations.}$$

The input and output requirements of this algorithm are as follows:

> *Input: The integers m, y, and a string representing the length-L computation of n. Also, the algorithm requires as input an integer A, which is an approximation to the number d of base-b digits of n. This approximation need only satisfy*
>
> $$\frac{1}{2} \;<\; \frac{A}{d} \;<\; 2.$$
>
> *We further restrict m so that $m < 2A + y + 1$, since otherwise $\gamma = 0$ satisfies the conclusion of our Theorem.*
>
> *Output: The level-y approximation to ν, encoded as a string of $y+1$ base-b digits.*

This theorem requires a little more explanation. First of all, the input to the algorithm will be a string of $O(L\log L + \log m + \log y)$ bits, which is smaller than the space requirement listed above (when $C > 1$). The $L\log L$, $\log m$, and $\log y$ terms here account for the number of bits needed to specify the length L computation of n, the index m, and the level y, respectively. The approximation A to d requires only $O(\log\log n)$ bits of space, and this turns out to be $O(L\log L)$: To see why this is so, we note that any length L computation produces an integer $n < 2^{2^{L-1}}$, which can be proved by induction. It follows then that $\log\log n = O(L)$.

The output of the algorithm will be a string of $O(y)$ bits, representing the base-b approximation γ to ν. The number γ will have only $y+1$ base-b digits, and will satisfy

$$|\gamma - \nu| \;<\; \frac{1}{b^y}.$$

Perhaps the most surprising aspect of the above theorem is that the indicated algorithm can require significantly less memory to find the approximation to ν than it does to write down the number n, which will have $O(\log n)$ base-b digits. The following corollary of the above theorem will make this point clear:

Corollary 1 *Suppose that $a, b \geq 2$ and $m, t, y \geq 1$ are all integers. There exists an algorithm which computes a level-y base-b approximation γ to ν, where*

$$\nu \;=\; \left\{ \frac{a^t}{b^m} \right\}.$$

This algorithm requires only

$$O(yt^{2/3} \log^C(a+b+y+m+t)) \;\; \text{bits of memory,}$$

where $C > 0$, and performs

$$O(yt \log^C(a+b+y+m+t)) \;\; \text{bit operations.}$$

Now, the number of bits needed to write down the number $n = a^t$ is clearly $O(t \log a)$; and yet, if, say, we take $y = 1$, this algorithm requires only $t^{2/3+o(1)}$ bits of space.

This corollary follows since a^t has computational complexity

$$L \;=\; O(\log^2(a+t)).$$

To see this, we note that a^t can be generated by repeated squaring: If $t = 2^{t_1} + \cdots + 2^{t_s}$, then $a^t = a^{2^{t_1}} \cdots a^{2^{t_s}}$. These numbers a^{2^h} can be computed by starting with a; then squaring to get a^2; then squaring again to get a^4; then continuing, this produces the list $a, a^2, a^4, ..., a^{2^h}$ after only h multiplications.

The rest of this paper is organized as follows: In the next section we will state the Main Theorem (Theorem 2) and then use it to deduce Theorem 1. In section 3 we give a proof of the Main Theorem. Finally, in section 4 we give a proof of a proposition (Proposition 1), which is an auxillary result needed for the proof of the Main Theorem.

2 Main Theorem and Proof of Theorem 1

Theorem 1 is actually a corollary of a more general result concerning approximations to ν. In this section we will state this result, which will henceforth be called the Main Theorem, and then show how to apply it to prove Theorem 1. The proof of this theorem, as well as a brief description of the ideas used to prove it, can be found in section 3; also, at the end of Section 3.1, we will give a brief statement on how to parallelize the algorithm.

We suppose that $b \geq 2$ is an integer, which is to be the base used; that $0 < n/a^t < 1$ is some rational number where $n, a \geq 1$, $t \geq 0$ are integers, and where n has computational complexity L; that $y \geq 1$ is some level of precision to be used; and finally, that $\mu \geq 0$ is some integer. Then, given any pair of integers S, T satisfying

$$ST \ > \ \frac{3(\log n + (\mu + y + 2)\log b)}{\log a}, \ a^S \ > \ T^2, \tag{2}$$

we have the following

Theorem 2 (Main Theorem) *Let*

$$\nu \ = \ \left\{ b^\mu \frac{n}{a^t} \right\}. \tag{3}$$

There exists an algorithm which computes a level-y approximation γ to this number ν, where the space and time requirements of the algorithm are as follows:

> *Space: $O(yL(S+T)\log^C(y + L + S + T + a + b + \mu + \log t))$ bits of memory, where $C > 0$.*
>
> *Time: $O(yL(ST + T^3)\log^C(y + L + S + T + a + b + \mu + \log t))$ bit operations.*

Here we give more precise information about the input and output specifications of the algorithm:

> Input: The positive integers a, t, μ, y, S and T, as well as a string of $O(L \log L)$ characters representing the length-L computation needed to produce n.

Output: The algorithm will give an approximation γ to ν. This approximation will have $y+1$ base-b digits, and will satisfy

$$|\gamma - \nu| \ < \ \frac{1}{b^y}.$$

To prove Theorem 1, using this result, we let $a = b$, $t = 2A$, and $\mu = t - m$. We note that this gives

$$\frac{b^\mu n}{a^t} \ = \ \frac{n}{b^m}.$$

We also let

$$\begin{aligned} S &= \left\lfloor \left(\frac{3(\log b)(3A+\mu+y+2)}{\log a} \right)^{2/3} \right\rfloor + 1, \\ T &= \left\lfloor \left(\frac{3(\log b)(3A+\mu+y+2)}{\log a} \right)^{1/3} \right\rfloor + 1. \end{aligned}$$

We note that this choice of S and T satisfies (2). [1]

Now, applying the algorithm described in Theorem 2 with the parameters indicated above, we get the same output as described in Theorem 1. The running time and space requirements to run this algorithm are also as stated in Theorem 1 for our particular choices of S and T.

3 Proof of Theorem 2

Let S and T be as in (2), and let r and k be integers such that

$$t \ = \ Sk - r, \ 0 \leq r \leq S-1.$$

Then, we have that

$$\alpha \ = \ \frac{n}{a^t} \ = \ \frac{na^r}{a^{Sk}}.$$

The idea of the proof of Theorem 2 is to approximate $b^\mu \alpha$ (and therefore ν) as follows:

$$b^\mu \alpha \ = \ \gamma_1 + \cdots + \gamma_T + E, \tag{4}$$

[1] To show this, one needs the fact that $3A \log b > \log n$, which follows since n has $\leq 2A$ base-b digits.

where

$$\gamma_j \;=\; b^{\mu}\frac{A_j}{a^S - j},$$

for some rationals $A_1, ..., A_T$, and where

$$|E| \;<\; \frac{1}{b^{y+2}} \text{ for } ST \text{ sufficiently large.} \tag{5}$$

Then, we will find approximations $\gamma'_1, ..., \gamma'_T$ to $\{\gamma_1\}, ..., \{\gamma_T\}$. Now, if the precision of these approximations $\gamma'_1, ..., \gamma'_T$ is high enough, and if we we let Σ satisfy

$$\Sigma \;=\; \{\gamma'_1 + \gamma'_2 + \cdots + \gamma'_T\},$$

then Σ will be an approximation to ν; and, if we then take γ to be the closest number to Σ having $y+1$ base-b digits, then γ will be a level-y approximation to ν.

We claim that the approximations $\gamma'_1, ..., \gamma'_T$ to $\{\gamma_1\}, ..., \{\gamma_T\}$ need only have

$$w \;=\; y \;+\; \left\lfloor \frac{\log T}{\log b} \right\rfloor \;+\; 3,$$

base-b digits (and be level-w approximations), in order to guarantee that Σ is a level-$y+1$ approximation to ν. Note that this would imply that

$$|\{\gamma_j\} \;-\; \gamma'_j| \;<\; \frac{1}{Tb^{y+2}}.$$

To see only w base-b digits are needed, we note that if these numbers γ'_j satisfy this last inequality, then by the triangle inequality,

$$\begin{aligned} |\gamma - \nu| \quad &\leq \quad \frac{1}{b^{y+1}} + |\Sigma - \nu| \;\leq\; \frac{1}{b^{y+1}} + \sum_{j=1}^{T} |\gamma'_j - \{\gamma_j\}| \;+\; |E| \\ &< \quad \frac{1}{b^{y+1}} + \frac{1}{b^{y+2}} + \frac{1}{b^{y+2}} \;\leq\; \frac{1}{b^y}, \end{aligned}$$

as claimed.

Let us now find a set of values for $A_1, ..., A_T$ which make (4) hold: Using the geometric series identity, we have that

$$\sum_{i=1}^{T} \frac{A_i}{a^S - i} \;=\; \sum_{j=1}^{\infty} \frac{B_j}{a^{Sj}}, \tag{6}$$

where

$$B_j \;=\; A_1 + A_2 2^{j-1} + A_3 3^{j-1} + \cdots + A_T T^{j-1}.$$

We seek values for $A_1, ..., A_T$ so that

$$B_j \;=\; \begin{cases} 0, & \text{for } 1 \le j \le T,\ j \ne k; \\ na^r, & \text{for } j = k. \end{cases}$$

The following Proposition gives the solution we seek

Proposition 1 *We have that*

$$A_j \;=\; \frac{na^r \left(\text{Coef. of } x^{k-1} \text{ in } \prod_{\substack{h=1 \\ h \ne j}}^{T} x - h \right)}{\prod_{\substack{h=1 \\ h \ne j}}^{T} j - h}; \tag{7}$$

and,

$$|A_j| \;\le\; nTa^S 4^T. \tag{8}$$

Note that this implies

$$\gamma_j \;=\; b^\mu \frac{(-1)^{T-j} na^r \left(\text{Coef. of } x^{k-1} \text{ in } \prod_{\substack{h=1 \\ h \ne j}}^{T} x - h \right)}{(j-1)!(T-j)!(a^S - j)}.$$

From this proposition we deduce that

$$\begin{aligned} |E| &= b^\mu \left| \sum_{j=T+1}^{\infty} \frac{A_1 + A_2 2^{j-1} + \cdots + A_T T^{j-1}}{a^{Sj}} \right| \\ &\le b^\mu \sum_{j=T+1}^{\infty} \frac{(nTa^S 4^T) T^j}{a^{Sj}} \\ &= \frac{na^S 4^T T^{T+2} b^\mu}{a^{S(T+1)}} \sum_{j=0}^{\infty} \frac{T^j}{a^{Sj}} \\ &= \frac{na^S 4^T T^{T+2} b^\mu}{a^{ST}(a^S - T)} \;<\; \frac{nb^\mu}{a^{ST/3}} \;<\; \frac{1}{b^{y+2}}, \end{aligned}$$

for ST large enough; and so, (5) follows.

We now have all the ingredients necessary to prove Theorem 2, which we will give as the following algorithm:

3.1 Algorithm 1

The input, output, and requirements of this algorithm are as stated in Theorem 2. Here are the steps of the algorithm:

1. Let

$$w \;=\; y + \left\lfloor \frac{\log T}{\log b} \right\rfloor + 3.$$

Note that this choice of w satisfies

$$\frac{T}{b^w} \;\leq\; \frac{1}{b^{y+2}}.$$

2. Set $\Sigma = 0$, and let r, S, T and k be as described at the beginning of this section.

3. For j from 1 to T do steps 4 through 8.

4. Compute

$$Q \;\leftarrow\; a^S - j.$$

5. Set

$$v \;\leftarrow\; (j-1)!(T-j)!Q.$$

This number can be computed using $O((T+S)\log^C(T+S))$ bit operations, and just as much memory (for some $C > 0$).

6. Apply Algorithm 2 (given in the next subsection of the paper) to compute

$$H \;\leftarrow\; \text{Coef. of } x^{k-1} \text{ in } \prod_{\substack{h=1 \\ h \neq j}}^{T} x - h.$$

This step requires $O(T^2 \log^C T)$ bit operations and $O(T \log^C T)$ bits of memory.

7. Compute

$$u \;\leftarrow\; (-1)^{T-j} n a^r H \pmod{v},\; 0 \leq u \leq v-1.$$

Since n has computational complexity L, this step requires only $O(L(S+T)\log^C(L+S+T+a))$ bit operations, and just as much memory.

8. Find a number τ having $w+1$ base-b digits satisfying

$$|\tau - \phi| \ < \ \frac{1}{b^w},$$

where

$$\phi \ = \ \left\{ b^\mu \frac{u}{v} \right\}.$$

(So, τ will be a level-w approximation to ϕ.)

We note that this number τ can be easily computed by first letting

$$u_0 \ \equiv \ b^\mu u \pmod{v}, \ 0 \le u_0 \le v-1,$$

and then noting that

$$\left\{ \frac{b^\mu u}{v} \right\} \ = \ \frac{u_0}{v} \ = \ (0.r_{-1}r_{-2}...)_b.$$

Then, by finding the first $w+1$ significant digits of u_0/v, and letting $\tau = (0.r_{-1}...r_{-w-1})_b$ one see that the above inequalities are satisfied.

9. Set

$$\Sigma \ \leftarrow \ \{\Sigma + \tau\}.$$

We only need to do level-w arithmetic in base-b here.

(If $j < T$, then increment j and loop back to step 4.)

10. (We assume $j = T$.) Let γ be the number having $y+1$ base-b digits which comes nearest to Σ, and then OUTPUT γ.

We note that we can perform the operations in steps 3 through 8, with different values of j, in parallel. For example, given two processors, we can assign processor 1 to perform steps 3 through 8, with values of $j \le T/2$, and then assign processor 2 to do the same, but with $T/2 < j \le T$. This would result in an two-fold reduction in the running time, as long as μ is sufficiently large. Of course, the memory requirements would double, because each of the two processors would require their own seperate memories.

More generally, we have that, given M processors, for μ sufficiently large, Algorithm 1 can be computed in parallel, resulting in an M-fold reduction in running time, but an M-fold increase in memory requirements.

3.2 Algorithm 2

Input: T, k, j.

Output: Coef. of x^{k-1} in $\prod_{\substack{h=1 \\ h \neq j}}^{T} x - h$.

Requirements: The algorithm performs $O(T^2 \log^C T)$ bit operations (for some $C > 0$), but requires only $O(T \log^C T)$ bits of memory.

1. Let P be the least integer such that

$$\Delta \;=\; \prod_{\substack{p \leq P \\ p \text{ prime}}} p \;\geq\; 2^{T+1} T!$$

Note: $P = O(T \log T)$, and can be computed using $O(T \log^D T)$ bit operations (for some $D > 0$); and so, we can compute and store P within the time and space requirements listed above for the algorithm. We also note that every coefficient of the polynomial in the output specifications is less than $\Delta/2$ in absolute value.

2. Set $\Sigma = 0$.

3. For each prime $p \leq P$ do steps 4 through 8.

4. Compute the polynomial

$$f(x) \;\equiv\; \prod_{\substack{h=1 \\ h \neq j}}^{T} x - h \pmod{p}.$$

Note: This polynomial can be stored as a length-T coefficient vector, and the number of bits required to store such a vector is $O(T \log p) = O(T \log T)$; also, this polynomial can be computed using $O(T \log^D T)$ bit operations by making use of FFT's and a divide-and-conquer strategy for polynomial multiplication. The divide-and-conquer part of the algorithm can probably best be described as the following recursive procedure: First, we suppose that L is a set of polynomials to be producted together modulo p, and Product(L)

denotes the procedure for computing this product. The pseudocode for this procedure is given as follows:

If $|L| = 1$ (i.e. L has only one polynomial), then
    RETURN the contents of $L \pmod p$;
Else, if $|L| \geq 2$, say $L = \{f_1, ..., f_t\}$, then
    RETURN
        $\text{Product}(\{f_1, ..., f_{\lfloor t/2 \rfloor}\}) \cdot \text{Product}(\{f_{\lfloor t/2 \rfloor + 1}, ..., f_t\}) \pmod p$

Now, using FFT's to perform the polynomial multiplication in this second step (the 'Else' step), we see that if the two polynomials being multiplied together have degrees ℓ_1 and ℓ_2, respectively, then the multiplication should take no more than $O((\ell_1+\ell_2)\log^D(\ell_1+\ell_2+p))$ bit operations. Now, if we run Product(L) starting with L consisting of all linear factors $x-h$, $1 \leq h \leq T$, $h \neq j$, then if $T-1$ is a power of 2, the procedure products together $(T-1)/2$ pairs of degree 1 polynomials; $(T-1)/4$ pairs of degree 2 polynomials; and so on, all the way down to two polynomials of degree $(T-1)/2$. So, the total number of bit operations required to run this producedure is

$$\ll (\log^D T) \sum_{j \leq (\log T)/\log 2 + 1} \frac{T}{2^j} 2^j \;=\; O(T \log^D T).$$

The memory requirements (in bits) are likewise of the same order.

5. Set

$$H \;\leftarrow\; \text{Coef. of } x^{k-1} \text{ in } f(x) \pmod p.$$

6. Set

$$N \;\leftarrow\; (\Delta/p)^{-1} H \pmod p, \text{ where } 0 \leq N \leq p-1.$$

7. Set

$$\Sigma \;\leftarrow\; \Sigma + \frac{N\Delta}{p}.$$

8. Increment the value of p, and return to step 4, unless $p > P$, in which case we proceed to step 9.

9. Let r be the least residue in absolute value of $\Sigma \pmod{\Delta}$.

10. Return the value of r, and STOP.

It is relatively easy to see that the algorithm requires no more than the indicated space and time requirements.

The idea behind the algorithm is that we use the Chinese Remainder Theorem to compute the x^{k-1} coefficient of our polynomial, and the computation in step 7 is just an "on the fly" CRT calculation. This calculation is based on the following fact: If $q_1, ..., q_h$ are coprime, and if $a_1, ..., a_h$ are any integers, then if we set

$$\Delta' \;=\; \prod_{i=1}^{h} q_i,$$

and

$$\Sigma' \;=\; \sum_{i=1}^{h} b_i \frac{\Delta'}{q_i}, \text{ where } b_i \equiv a_i(\Delta'/q_i)^{-1} \pmod{q_i},$$

then

$$\Sigma' \;\equiv\; a_i \pmod{q_i}, \text{ for every } i = 1, 2, ..., h.$$

One might guess that the coefficient of our polynomial can be computed using less resources by using a "Fourier Series" method; that is,

$$\text{Coef. } x^{k-1} \text{ in } f(x) \;=\; \frac{1}{T} \sum_{\ell=0}^{T-1} e^{-2\pi i \ell(k-1)/T} f(e^{2\pi i \ell/T}).$$

It is not obvious (to me) how to do this without using the special form of the polynomial $f(x)$: First of all, we would need to maintain $\gg T$ digits of precision for each term in the sum, since any particular coefficient of the polynomial $f(x)$ can have size $2^{cT \log T}$, for some $c > 0$. Thus, $\gg T^2$ bit operations would be needed to compute each term $f(e^{2\pi i \ell/T})$. In total, $\gg T^3$ bit operations would be needed to evaluate all the terms in the sum. If one tries to use FFT's to evaluate *all* the terms in the sum at the same time, this reduces the running time to $O(T^2 \log^D T)$ bit operations; however, the memory requirements then increase to $\gg T^2$ bits of storage, which is the

amount needed to store the all numbers $f(e^{2\pi i\ell/T})$, $0 \leq \ell \leq T$ to $\gg T$ bits of precision. Even if we try a discrete version of this method, where the polynomials are computed, say, modulo 2^k for $k \gg T$, and the roots of unity are roots of unity modulo 2^k, we would run into the same difficulties.

4 Proof of Proposition 1

The A_i's can be computed by solving the equation

$$\begin{pmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & 2 & 3 & \cdots & T \\ 1 & 2^2 & 3^2 & \cdots & T^2 \\ \vdots & \vdots & \vdots & \cdots & \vdots \\ 1 & 2^{T-1} & 3^{T-1} & \cdots & T^{T-1} \end{pmatrix} \begin{pmatrix} A_1 \\ A_2 \\ A_3 \\ \vdots \\ A_T \end{pmatrix} = \begin{pmatrix} 0 \\ \vdots \\ 0 \\ na^r \\ 0 \\ \vdots \\ 0 \end{pmatrix}. \tag{9}$$

If we call the matrix on the left-hand-side M, then

$$A_j \;=\; na^r M^{-1}_{j,k}, \tag{10}$$

where $M^{-1}_{j,k}$ is the entry in the jth row, kth column of M^{-1}.

We will calculate $M^{-1}_{j,k}$ via polynomial interpolation: We have that for any set of ordered pairs

$$(1, b_1),\ (2, b_2),\ ...,\ (T, b_T),$$

where $b_1, ..., b_T \in \mathbb{C}$, there exists a unique degree $T-1$ polynomial $f(x) \in \mathbb{C}[x]$ such that

$$f(i) \;=\; b_i,\ \text{for all } i = 1, 2, ..., T;$$

moreover, if we write

$$f(x) \;=\; c_T x^{T-1} + c_{T-1}x^{T-2} + \cdots + c_2 x + c_1,$$

then these coefficients c_i can be calculated in two different ways: The first way is through basic linear algebra, since

$$\begin{pmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & 2 & 2^2 & \cdots & 2^{T-1} \\ 1 & 3 & 3^2 & \cdots & 3^{T-1} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & T & T^2 & \cdots & T^{T-1} \end{pmatrix} \begin{pmatrix} c_1 \\ c_2 \\ c_3 \\ \vdots \\ c_T \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_T \end{pmatrix}. \tag{11}$$

We notice that the matrix on the left-hand-side is M', the transpose of our matrix M.

The second way of calculating the c_i's is by Lagrange interpolation, which gives

$$f(x) \;=\; \sum_{i=1}^{T} b_i \prod_{\substack{h=1 \\ h\neq i}}^{T} \frac{x-h}{i-h}. \tag{12}$$

Now, if we suppose that

$$b_i \;=\; \begin{cases} 0, & \text{if } i \neq j, \\ 1, & \text{if } i = j, \end{cases}$$

then for this choice of b_i's, one sees from (11) that

$$c_k \;=\; (M')^{-1}_{k,j} \;=\; M^{-1}_{j,k}.$$

On the other hand, from (12) we see that

$$c_k \;=\; \text{Coef. of } x^{k-1} \text{ in } \prod_{\substack{h=1 \\ h\neq j}}^{T} \frac{x-h}{j-h}.$$

Thus,

$$M^{-1}_{j,k} \;=\; \text{Coef. of } x^{k-1} \text{ in } \prod_{\substack{h=1 \\ h\neq j}}^{T} \frac{x-h}{j-h},$$

and we conclude from this and (10) that (7) holds.

Finally, to prove (8), we note that

$$\left| \prod_{\substack{h=1 \\ h\neq j}}^{T} j-h \right| \;=\; (T-j)!(j-1)!,$$

The coefficient of x^{k-1} in the above polynomial is clearly less than

$$T!\binom{T}{k-1} \;<\; T!2^T.$$

So,

$$|A_j| \;\leq\; na^S 2^T \frac{T!}{(T-j)!(j-1)!} \;=\; jna^S 2^T \binom{T}{j} \;<\; nTa^S 4^T,$$

which proves (8).

5 Acknowledgements

I would like to thank Richard Hudson for an email he sent to me, which got me interested in these digit calculation questions, which eventually lead me to prove the theorems listed above. I would also like to thank Kevin Hare for pointing out to me that my algorithm above is highly parallelizable.

References

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

[2] M. Shub and S. Smale, On the Intractability of Hilbert's Nullstellensatz and an Algebraic Version of "P=NP?", *Duke Math. Jour.* **81** (1995), 47-54.