Things about computing I needed to learn over the years ...

Bradley J. Lucier

The series of HAKMEMs (Hackers Memos) were produced at MIT beginning in the 1960s and concerned themselves with how to compute things efficiently, but mainly at the word level, using bit operations and other low-level techniques.

I'm starting this document to bring together high-level techniques, or just statements that summarize things I've had to learn over the years, sometimes the hard way. Knuth's "Seminumerical Algorithms", volume II of his "Art of Computer Programming", has been in print for 35 years. It would be nice to bring together some techniques that build on that material. Similarly, fast and accurate computational methods for ordinary differential equations and partial differential equations have been developed over the past 25 years: multigrid, multipole methods, symplectic integrators. All these techniques offer vast improvements over what was used previously.

I hope other people will continue this text or correct the things that are wrong. I'll find an free license to publish it under.

Computing with integers: How it's done. Computers work on what are called integer "words" of various sizes—these are the objects they can add, subtract, multiply, etc., in a single instruction. Large integers are represented as coefficients of powers of the word. If a word is 2^k for example, then a large positive integer a would be written as

$$a = \sum_{j=1}^{n_a} a_j (2^k)^j,$$

where $0 \le a_j < 2^k$ and n_a is the number of words necessary to represent a. Negative numbers are represented in a number of ways, the most common being sign-magnitude

$$a = (-1)^{e_a} \sum_{j=1}^{n_a} a_j (2^k)^j$$

where $e_a = 0$ if a is positive and $e_a = 1$ if a is negative, or twos-complement

$$a = \sum_{j=1}^{n_a} a_j (2^k)^j - 2^{k \times n_a} \operatorname{topbit}(a_{n_a}),$$

where $\operatorname{topbit}(d) = 1$ if $d \ge 2^{k-1}$ and zero otherwise.

Adding and subtracting such numbers is easy—it takes $O(\max(n_a, n_b))$ operations to add two numbers a and b. The naive multiplication one learns in school takes $O(n_a \times n_b)$ operations to compute $a \times b$, which is quite time-consuming when a and b are large. (You begin to notice it when a and b have a few thousand bits.) If a is small (fits in a word, say) and b is large, then there's really no faster way to do it—it takes $O(b_n)$ operations to calculate $a \times b$. Multiplying by 2^k is easy, it reduces to a shift and takes $O(\max(n_a, k))$ operations.

Note that if $a = b$, then the number of word operations to calculate a^2 can be can be cut roughly in half.

If a and b are large then some useful techniques come into play. The simplest is Karatsuba's decomposition: if we have $a, b \approx 2^{2m}$ and write

$$a = a_1 \times 2^m + a_0,\ b = b_1 \times 2^m + b_0 \text{ with } 0 \leq a_i, b_i < 2^m$$

then

$$a \times b = (a_1 \times b_1) \times 2^{2m} + [a_1 \times b_1 - (a_1 - a_0) \times (b_1 - b_0) + a_0 \times b_0] \times 2^m + a_0 \times b_0.$$

Note that we've reduced multiplying two $2m$-bit numbers to the problem of three multiplications of two m-bit numbers (which is still expensive) and four additions/subtractions (which is not). This reduces the complexity of the multiplication when $n_a \approx n_b$ to $O(n_a^{\log 3/\log 2}) < O(n_a^2)$ operations (see Knuth, *Seminumerical Algorithms*). Again, note that some simplification is possible if $a = b$, since all the suboperations are squares. Once the subparts get small enough, you use the naive algorithm again.

If you get really tricky, then you can use Fast Fourier Transforms to multiply a and b; it then takes $O(n_a \log n_a)$ operations to calculate $a \times b$ when $n_a \approx n_b$. Again, see Knuth. Again, one can cut the time somewhat when computing a^2.

If either the quotient or the divisor of the division are relatively small ($<$ a few thousands bits, say), then the grade-school long division algorithm is still the best. Fast algorithms based on Newton's method are available for integer division and integer square root, where, given $a > 0$ and $b > 0$, we want to find $0 \leq q$, $0 \leq r < b$, and $d \geq 0$ such that

$$a = qb + r \text{ and } d^2 \leq a < (d+1)^2.$$

See Knuth for the division algorithm, and the Gambit-C source code for the square root algorithm.

In each case, one gets that the number of word operations needed to divide a $2m$-bit number by an m-bit number, or to take the square root of a $2m$-bit number, is bounded by a constant times the number of word operations needed to multiply two m-bit numbers. Since Gambit-C implements versions of all these tricky algorithms, we can find out approximately what that constant is for Gambit-C running on a 2GHz PowerPC 970:

```
> (define a (expt 3 1000000))              ; calculate $3^{1,000,000}$
> (define b (expt 3 1000001))
> (define c (time (* a a)))                ; time for a square
(time (* a a))
    223 ms real time
    220 ms cpu time (220 user, 0 system)
    2 collections accounting for 9 ms real time (10 user, 0 system)
    17871352 bytes allocated
    no minor faults
    no major faults
> (define d (time (* a b)))                ; time for a multiply
(time (* a b))
    320 ms real time
```

```
    310 ms cpu time (310 user, 0 system)
    1 collection accounting for 13 ms real time (10 user, 0 system)
    21364376 bytes allocated
    no minor faults
    no major faults
> (define e (time (quotient c a)))          ; time for a divide
(time (quotient c a))
    1376 ms real time
    1390 ms cpu time (1390 user, 0 system)
    17 collections accounting for 94 ms real time (90 user, 0 system)
    107019976 bytes allocated
    no minor faults
    no major faults
> (define f (time (##exact-int.sqrt c)))    ; time for a sqrt
(time (##exact-int.sqrt c))
    2616 ms real time
    2610 ms cpu time (2610 user, 0 system)
    40 collections accounting for 213 ms real time (280 user, 0 system)
    225066744 bytes allocated
    no minor faults
    no major faults
```

There are two other important operations on nonnegative integers—the greatest common divisor (GCD) of a and b and a^b. GCD can be implemented as

```
(define (gcd x y)
  (if (= y 0)
      x
      (gcd y (remainder x y))))
```

GCD can require many large remainder computations, so it can be very slow. GCD is slowest when `x` and `y` are consecutive Fibonacci numbers, for which

$$\texttt{remainder}(F_n, F_{n-1}) = F_{n-2}.$$

So GCD(F_n, F_{n-1}) takes about n remainders, which takes as long as n divisions. For example, we can define F_n by

```
(define (fib n)   ; works for $n\geq 2$
  (let loop ((i       2)
             (fib_i-1 1)
             (fib_i   1))
    (if (= i n)
        fib_i
        (loop (+ i 1) fib_i (+ fib_i-1 fib_i)))))
```

With this code we get the following timings:

```
> (define a (fib 100000))
> (define b (fib 100001))
> (time (gcd a b))
(time (gcd a b))
    13013 ms real time
    12900 ms cpu time (12900 user, 0 system)
    2894 collections accounting for 3620 ms real time (3450 user, 0 system)
    1313462832 bytes allocated
    no minor faults
    no major faults
1
```

By contrast, $2^{100,000}$ and $3^{100,000}$ are quite a bit bigger than $F_{100,000}$ and $F_{100,001}$, yet we have

```
> (define a (expt 3 100000))
> (define b (expt 2 100000))
> (time (gcd a b))
(time (gcd a b))
    11000 ms real time
    11060 ms cpu time (11060 user, 0 system)
    2370 collections accounting for 2996 ms real time (3150 user, 0 system)
    1096108560 bytes allocated
    no minor faults
    no major faults
1
```

You can see that GCD can take a long time, longer than quotient or sqrt. For our usual test integers we have:

```
> (define a (expt 2 1000000))
> (define b (expt 3 1000000))
> (time (gcd a b))
(time (gcd a b))
    1288572 ms real time
    1225680 ms cpu time (1225680 user, 0 system)
    94755 collections accounting for 287789 ms real time (273560 user, 0 system)
    107397872656 bytes allocated
    no minor faults
    no major faults
1
```

(Sometime later ...) Someone on the comp.lang.scheme newsgroup pointed out that one can use

$$\mathrm{GCD}(\alpha \times 2^j, \beta \times 2^k) = 2^{\min(j,k)}\,\mathrm{GCD}(\alpha, \beta)$$

to speed up the computation of the GCD for integers with large powers of 2 in their factorization. With this optimization, we have

```
> (define a (expt 2 1000000))
> (define b (expt 3 1000000))
> (time (gcd a b))
(time (gcd a b))
    0 ms real time
    0 ms cpu time (0 user, 0 system)
    no collections
    80 bytes allocated
    no minor faults
    no major faults
1
```

This optimization speeds up the conversion of floating-point numbers to exact rationals by about a factor of two in Gambit-C.

After one abandons the naive algorithm for exponentiation based on the recursion

$$a^b = a \times (a^{b-1}),$$

there are two natural ways to implement exponentiation

```
(define (expt1 a b)
  (define (square x) (* x x))
```

```
  (cond ((= b 0) 1)
        ((even? b)
         (square (expt1 a (quotient b 2))))
        (else
         (* a (square (expt1 a (quotient b 2)))))))
(define (expt2 a b)
  (define (square x) (* x x))
  (cond ((= b 0) 1)
        ((even? b)
         (expt2 (square a) (quotient b 2)))
        (else
         (* a (expt2 (square a) (quotient b 2))))))
```

based on the recursions

$$a^b = \begin{cases} 1, & b = 0 \\ (a^{b/2})^2, & b \text{ even, and} \\ (a^{[b/2]})^2 \times a, & b \text{ odd,} \end{cases} \quad \text{and} \quad a^b = \begin{cases} 1, & b = 0 \\ (a^2)^{b/2}, & b \text{ even, and} \\ (a^2)^{[b/2]} \times a, & b \text{ odd,} \end{cases}$$

respectively, where $[x]$ is the largest integer no greater than x. The first formula saves all the squarings for last, when the numbers are biggest; the second one does squaring on the smallest numbers. Remember, squaring integers is faster than multiplying two nonidentical numbers. One can see a difference in the timings:

```
> (define a (time (expt1 3 1000000)))
(time (expt1 3 1000000))
    208 ms real time
    200 ms cpu time (200 user, 0 system)
    5 collections accounting for 32 ms real time (40 user, 0 system)
    17846656 bytes allocated
    no minor faults
    no major faults
> (define b (time (expt2 3 1000000)))
(time (expt2 3 1000000))
    1281 ms real time
    1320 ms cpu time (1320 user, 0 system)
    13 collections accounting for 69 ms real time (80 user, 0 system)
    92484064 bytes allocated
    no minor faults
    no major faults
> (= a b)
#t
> (define a (expt 3 1000000))
> (define b (time (* a a)))
(time (* a a))
    244 ms real time
    200 ms cpu time (200 user, 0 system)
    2 collections accounting for 17 ms real time (20 user, 0 system)
    17871352 bytes allocated
    no minor faults
    no major faults
```

Multiplication and division by numbers divisible by a power of 2 can be computed more quickly, because multiplication by a power of 2 can be implemented with shifts, which take time proportional to the size of the result. For example if $a = \alpha 2^j$ and $b = \beta 2^k$ for odd α and β, we have

$$a \times b = (\alpha \times \beta) 2^{k+j}.$$

If α and β are much shorter than a and b, then the latter multiplication, using the usual tricks, can be executed much more quickly that $a \times b$.

Division is a bit more tricky. If we have $b = \beta \times 2^k$ and $a = \alpha \times 2^k + r$, $0 \le r < 2^k$ and

$$\alpha = q_0\beta + r_0, \quad 0 \le r_0 < \beta$$

then

$$a = 2^k\alpha + r = 2^k q_0\beta + 2^k r_0 + r = q_0 \times b + 2^k r_0 + r.$$

Now

$$2^k r_0 + r < 2^k(r_0 + 1) \le 2^k\beta = b,$$

so the remainder when dividing a by b is $2^k r_0 + r$ and the quotient of a by b is truly q_0. Gambit-C applies this optimization to when $2^{2k} \ge b$.

Without this optimization we have

```
> (define a (expt 10 10000000))
> (define b (expt 2 10000000))
>  (define c (time (quotient a b)))
(time (quotient a b))
    539 ms real time
    480 ms cpu time (480 user, 0 system)
    7 collections accounting for 90 ms real time (80 user, 0 system)
    83784128 bytes allocated
    no minor faults
    no major faults
```

while with it we have

```
> (define a (expt 10 10000000))
> (define b (expt 2 10000000))
> (define c (time (quotient a b)))
(time (quotient a b))
    28 ms real time
    20 ms cpu time (20 user, 0 system)
    1 collection accounting for 2 ms real time (10 user, 0 system)
    6652648 bytes allocated
    no minor faults
    no major faults
```

To summarize: For any integers a and b, the number of operations required are $O(\max(n_a, n_b))$ for addition and subtraction; $O(n_a \times n_b)$ for multiplication when at least one of a and b are small (< 1000 bits, say) and $O(n_a \log n_a)$ when $n_a \approx n_b$ are large; squaring has the same complexity as multiplication, but a smaller constant; division and square roots take a constant multiple of multiplication; exponentiation takes about as long as the final squaring; GCD is quite expensive.

Computing with rationals: How it's done. Rationals are represented in the usual way, p/q with $\mathrm{GCD}(p, q) = 1$, and the usual arithmetic operations are also done in the usual way:

$$\frac{p}{q} \times \frac{r}{s} = \frac{pr/(\mathrm{GCD}(pr, qs))}{qs/(\mathrm{GCD}(pr, qs))} \quad \text{and} \quad \frac{p}{q} \pm \frac{r}{s} = \frac{(ps \pm rq)/(\mathrm{GCD}(ps \pm rq, qs))}{qs/(\mathrm{GCD}(ps \pm rq, qs))}.$$

No GCD is necessary to compute and normalize $(p/q)^2$. GCDs on large integers are expensive, so we rewrite the first formula as

$$\frac{p}{q} \times \frac{r}{s} = \frac{(p/(\mathrm{GCD}(p,s)) \times (r/\mathrm{GCD}(r,q))}{(q/\mathrm{GCD}(r,q)) \times (s/(\mathrm{GCD}(p,s))},$$

since we know that $\mathrm{GCD}(p,q) = \mathrm{GCD}(r,s) = 1$.

A somewhat extreme example of the difference between the first and second formulas is given below; with the first, we get

```
> (define a (/ (expt 2 10000) (expt 3 10000)))
> (define b (/ (expt 3 10000) (expt 5 10000)))
> (define c (time (* a b)))
(time (* a b))
    431 ms real time
    410 ms cpu time (410 user, 0 system)
    278 collections accounting for 98 ms real time (100 user, 0 system)
    48577608 bytes allocated
    no minor faults
    no major faults
```

while with the second we get

```
> (define a (/ (expt 2 10000) (expt 3 10000)))
> (define b (/ (expt 3 10000) (expt 5 10000)))
> (define c (time (* a b)))
(time (* a b))
    115 ms real time
    110 ms cpu time (110 user, 0 system)
    115 collections accounting for 34 ms real time (10 user, 0 system)
    19227216 bytes allocated
    no minor faults
    no major faults
```

Gambit-C uses the second formula.

Exponentiation of rationals should use the formula

$$\left(\frac{p}{q}\right)^n = \frac{p^n}{q^n}$$

to avoid the unneeded GCDs to normalize the result. Inversion of rationals and addition and subtraction between rationals and integers need no GCDs:

$$x \pm \frac{p}{q} = \frac{xq \pm p}{q}.$$

So we come up with another way to compute the nth Fibonacci number:

```
(define (fib-ratio n)
  (if (= n 1)
      1
      (+ 1 (/ (fib-ratio (- n 1))))))
(define (fib n)
  (numerator (fib-ratio n)))
```

which gives

```
> (time (fib 1000))
(time (fib 1000))
    4 ms real time
    10 ms cpu time (10 user, 0 system)
    1 collection accounting for 2 ms real time (0 user, 0 system)
    363544 bytes allocated
    no minor faults
    no major faults
70330367711422815821835254877183549770181269836358732742604905087154537118196 9
33579742249494562611733487750449241765991088186363265450223647106012053374121 2
73867339111198139373125598767690091902245245323403501
```

Computing sums and products: Binary splitting. A simple program to calculate $n!$ is given by

```
(define (factorial n)
  (let loop ((i 1)
             (result 1))
    (if (> i n)
        result
        (loop (+ i 1)
              (* i result)))))
```

We can assume that n fits in a single word (or else we could not store the result), so each of the n multiplications `(* i result)` multiplies a single word by an increasingly large number; the execution time is at least quadratic in n:

```
> (define a (time (factorial 1000)))
(time (factorial 1000))
    4 ms real time
    10 ms cpu time (10 user, 0 system)
    no collections
    1027032 bytes allocated
    no minor faults
    no major faults
> (define a (time (factorial 10000)))
(time (factorial 10000))
    543 ms real time
    550 ms cpu time (550 user, 0 system)
    74 collections accounting for 152 ms real time (170 user, 0 system)
    70965288 bytes allocated
    no minor faults
    no major faults
> (define a (time (factorial 100000)))
(time (factorial 100000))
    102263 ms real time
    102040 ms cpu time (102040 user, 0 system)
    9252 collections accounting for 25011 ms real time (25060 user, 0 system)
    9039052936 bytes allocated
    no minor faults
    no major faults
```

If for some $1 < m < n$ we write $n!$ as

$$n! = \left(\prod_{i=1}^{m} i\right) \times \left(\prod_{i=m+1}^{n} i\right),$$

compute both subproducts separately and multiply them together, then we will be able to use the machinery for multiplying large numbers together much faster than quadratically in the size of the numbers. We can then apply this idea recursively to each of the subproducts, etc. If the number of the terms in the product is small enough, then we should just multiply them together without worrying about the recursion. If at each point we break the product into roughly equal numbers of terms, we have

```
(define (partial-factorial m n)
  ;; computes the product (m+1) * ... * (n-1) * n
  (if (< (- n m) 10)
      (do ((i (+ m 1) (+ i 1))
           (result 1 (* result i)))
          ((> i n) result))
      (* (partial-factorial m (quotient (+ m n) 2))
         (partial-factorial (quotient (+ m n) 2) n))))
```

This results in significantly reduced computational time for large n:

```
> (define a (time (partial-factorial 0 1000)))
(time (partial-factorial 0 1000))
    6 ms real time
    10 ms cpu time (10 user, 0 system)
    1 collection accounting for 3 ms real time (0 user, 0 system)
    201112 bytes allocated
    no minor faults
    no major faults
> (define a (time (partial-factorial 0 10000)))
(time (partial-factorial 0 10000))
    55 ms real time
    60 ms cpu time (60 user, 0 system)
    5 collections accounting for 8 ms real time (20 user, 0 system)
    5018240 bytes allocated
    no minor faults
    no major faults
> (define a (time (partial-factorial 0 100000)))
(time (partial-factorial 0 100000))
    1209 ms real time
    1200 ms cpu time (1200 user, 0 system)
    91 collections accounting for 174 ms real time (210 user, 0 system)
    111991776 bytes allocated
    no minor faults
    no major faults
```

Similarly, one can split sums and remove common factors to work with smaller integers and rationals and hence speed computations. For example, we have

$$e \approx \sum_{k=0}^{n-1} \frac{1}{k!} = \sum_{k=0}^{m-1} \frac{1}{k!} + \sum_{k=m}^{n-1} \frac{1}{k!} = \sum_{k=0}^{m-1} \frac{1}{k!} + \frac{1}{m!} \sum_{k=m}^{n-1} \frac{1}{(m+1)(m+2)\cdots k},$$

and the error is a bit bigger than $1/n!$.

One can split each of the sums on the right hand side in a similar way, until we get to sums that are so small that it is more efficient to just compute them directly. So we write the following code:

```
(define (binary-splitting-partial-sum m n
                                      partial-term
```

```
                                    common-factor-ratio)
  ;; sums (partial) terms from m to n-1
  ;; (partial-term n m) is the term at n with the common factors of terms >= m removed
  ;; (common-factor-ratio m n) is the ratio of the common factor of terms >= n divided by
  ;; the common factors of terms >= m
  (if (< (- n m) 10)
      (do ((i m (+ i 1))
           (result 0 (+ result (partial-term m i))))
          ((= i n) result))
      (+ (binary-splitting-partial-sum m
                                       (quotient (+ m n) 2)
                                       partial-term
                                       common-factor-ratio)
         (* (common-factor-ratio m (quotient (+ m n) 2))
            (binary-splitting-partial-sum (quotient (+ m n) 2)
                                          n
                                          partial-term
                                          common-factor-ratio)))))
(define (binary-splitting-sum n partial-term common-factor)
  (binary-splitting-partial-sum 0 n partial-term common-factor))
(define (binary-splitting-compute-e n)
  (binary-splitting-sum n
                        (lambda (m n) (/ (partial-factorial m n)))
                        (lambda (m n) (/ (partial-factorial m n)))))
```

With these definitions we obtain

```
> (define a (time (binary-splitting-compute-e 1000)))
(time (binary-splitting-compute-e 1000))
    423 ms real time
    420 ms cpu time (420 user, 0 system)
    141 collections accounting for 164 ms real time (190 user, 0 system)
    61860008 bytes allocated
    no minor faults
    no major faults
> (exact->inexact a)
2.718281828459045
> (integer-length (partial-factorial 0 1000))
8530
> (define e (time (binary-splitting-compute-e 10000)))
(time (binary-splitting-compute-e 10000))
    52582 ms real time
    52670 ms cpu time (52670 user, 0 system)
    13326 collections accounting for 17987 ms real time (17980 user, 0 system)
    5857017928 bytes allocated
    no minor faults
    no major faults
> (integer-length (partial-factorial 0 10000))
118459
```

So we compute about 8500 bits of e in .42 seconds and over 100,000 bits of e in 52.670 seconds.

This can be compared with the naive algorithm for the first sum for the approximation to e.

```
(define (naive-compute-e n)
  (do ((k 0 (+ k 1))
       (sum 0 (+ sum (/ (partial-factorial 0 k)))))
      ((= k n) sum)))
> (define b (time (naive-compute-e 1000)))
(time (naive-compute-e 1000))
```

```
    58813 ms real time
    58900 ms cpu time (58900 user, 0 system)
    17730 collections accounting for 22228 ms real time (21970 user, 0 system)
    7693043896 bytes allocated
    no minor faults
    no major faults
> (= a b)
#t
```

One may think that if we computed a running product for each term then things might improve significantly, but we find

```
(define (still-naive-compute-e n)
  (do ((k    0 (+ k 1))
       (term 1 (/ term (+ k 1)))
       (sum 0  (+ sum term)))
      ((= k n) sum)))
> (define c (time (still-naive-compute-e 1000)))
(time (less-naive-compute-e 1000))
    57566 ms real time
    57480 ms cpu time (57480 user, 0 system)
    17658 collections accounting for 22222 ms real time (22460 user, 0 system)
    7606985264 bytes allocated
    no minor faults
    no major faults
> (= a c)
#t
```

Similar techniques can be used to compute π, even from the formula

$$\frac{\pi}{4} = 4\arctan\frac{1}{5} - \arctan\frac{1}{239}.$$

For we have

$$\arctan x = \sum_{n=0}^{\infty}(-1)^n\frac{x^{2n+1}}{2n+1} = \sum_{n=0}^{m-1}(-1)^n\frac{x^{2n+1}}{2n+1} + x^{2m+1}\sum_{n=m}^{\infty}(-1)^n\frac{x^{2n-2m}}{2n+1}$$

so using our binary-splitting technique we have

```
(define (binary-splitting-compute-atan n x)
  ;; here we just consider the common factor to be x^(2n+1)
  (* x    ;; common factor for all terms
     (binary-splitting-sum n
                           (lambda (m n) (/ (expt x (* 2 (- n m)))
                                            (* (if (odd? n) -1 1) ( + (* 2 n) 1))))
                           (lambda (m n) (expt x (* 2 (- n m)))))))
```

This is an alternating series with terms in decreasing absolute value, so the error has the same sign as and has absolute values less than the first omitted term. If we use n terms in the approximation to $\arctan 1/5$, then, because

```
> (/ (log (/ 239)) (log (/ 5)))
3.4027181226574648
```

we need about one-third as many terms in the approximation to $\arctan 1/239$. So we write

```
(define (binary-splitting-compute-pi n)
  (* 4 (- (* 4 (binary-splitting-compute-atan n 1/5))
```

```
          (binary-splitting-compute-atan (quotient (* n 10) 34) 1/239))))
```

With these definitions we get

```
> (exact->inexact (binary-splitting-compute-pi 100))
3.141592653589793
> (define pi-1000 (time (binary-splitting-compute-pi 1000)))
(time (binary-splitting-compute-pi 1000))
    913 ms real time
    930 ms cpu time (930 user, 0 system)
    304 collections accounting for 375 ms real time (410 user, 0 system)
    128265800 bytes allocated
    no minor faults
    no major faults
> (define pi-1000 (time (binary-splitting-compute-pi 10000)))
(time (binary-splitting-compute-pi 10000))
    63466 ms real time
    63540 ms cpu time (63540 user, 0 system)
    16514 collections accounting for 22288 ms real time (22640 user, 0 system)
    7170210536 bytes allocated
    no minor faults
    no major faults
> (integer-length (* (expt 5 20001) 20001))
46456
```

So here we computed over 46,000 bits of π in 63.5 seconds. To be fair to this method, we should really compile the code. Very high-precision methods for computing π can be based in more sophisticated series.

Computing in fixed-point arithmetic. If you want to compute π to a million digits, say, then it may be useful to use an iterative method that computes with numbers of the form

$$x = n\beta^{-k}$$

where k is fixed and n is an integer. This is a fixed-point representation with k base-β fractional digits. If $x = m\beta^{-k}$ and $y = n\beta^{-k}$ are two numbers represented in this way, then we can compute k-digit approximations to the usual arithmetic operations using the formulas

$$\begin{aligned} x \pm y &= m\beta^{-k} \pm n\beta^{-k} = (m \pm n)\beta^{-k}, \\ x \times y &= (m \times n)\beta^{-2k} \approx [(m \times n)/\beta^k]\beta^{-k}, \\ x/y &= (m/n) \approx [(m \times \beta^k)/n]\beta^{-k}, \\ \sqrt{x} &= \sqrt{m\beta^{-k}} \approx \left[\sqrt{m \times \beta^k}\right]\beta^{-k}, \end{aligned}$$

where $[z]$ is the integer part of z. We can write functions to implement these operations, where we compute and keep only the integer multiple of β^{-k}:

```
(define (fixed.+ x y)
  (+ x y))
(define (fixed.- x y)
  (- x y))
(define (fixed.* x y)
  (quotient (* x y) beta^k))
(define (fixed.square x)
```

```
  (fixed.* x x))
(define (fixed./ x y)
  (quotient (* x beta^k) y))
(define (fixed.sqrt x)
  (##exact-int.sqrt (* x beta^k)))
```

and functions to convert between regular numbers and fixed numbers:

```
(define (number->fixed x)
  (round (* x beta^k)))
(define (fixed->number x)
  (/ x beta^k))
```

Here we assume that `beta^k` is a global variable to be defined appropriately.

We can use these functions to compute a high-precision approximation to π using the Brent-Salamin iteration for π, adapted from code in Gambit-C's benchmark suite:

```
(define (pi-brent-salamin)
  (let ((one (number->fixed 1)))
    (let loop ((a one)
               (b (fixed.sqrt (quotient one 2)))
               (t (quotient one 4))
               (x 1))
      (if (= a b)
          (fixed./ (fixed.square a) t)
          (let ((new-a (quotient (fixed.+ a b) 2)))
            (loop new-a
                  (fixed.sqrt (fixed.* a b))
                  (fixed.- t (* x (fixed.square (fixed.- new-a a))))
                  (* 2 x)))))))
```

With these definitions we get the following timings:

```
> (define beta^k (expt 10 60))
> (define c (pi-brent-salamin))
> c
3141592653589793238462643383279502884197169399375105820970795
> (define beta^k (expt 10 10000))
> (define c (time (pi-brent-salamin)))
(time (pi-brent-salamin))
    1061 ms real time
    1040 ms cpu time (1040 user, 0 system)
    241 collections accounting for 234 ms real time (220 user, 0 system)
    137051936 bytes allocated
    no minor faults
    no major faults
> (define beta^k (expt 10 1000000))
> (define c-10 (time (pi-brent-salamin)))
(time (pi-brent-salamin))
    282664 ms real time
    281090 ms cpu time (281090 user, 0 system)
    1230 collections accounting for 14994 ms real time (15000 user, 0 system)
    16580654288 bytes allocated
    no minor faults
    no major faults
```

Using $\beta = 2$ should give some speedup, but then you need to add the time to convert the binary fixed-point answer to decimal digits:

```
> (define k (+ 1 (inexact->exact (round (* 1000000 (log 10) (/ (log 2)))))))
> k
```

```
3321929
> (define beta^k (expt 2 k))
> (define c-2 (time (pi-brent-salamin)))
(time (pi-brent-salamin))
    232626 ms real time
    228870 ms cpu time (228870 user, 0 system)
    1278 collections accounting for 13713 ms real time (13590 user, 0 system)
    14898725848 bytes allocated
    no minor faults
    no major faults
> (define c-2->10 (time (quotient (* c-2 (expt 10 1000000)) beta^k)))
(time (quotient (* c-2 (expt 10 1000000)) beta^k))
    1476 ms real time
    1480 ms cpu time (1480 user, 0 system)
    7 collections accounting for 83 ms real time (90 user, 0 system)
    80875976 bytes allocated
    no minor faults
    no major faults
> (define d (- c-10 c-2->10))
> d
-2035084
```

Using $\beta = 2$ didn't help so much. Out of a million digits, the two answers differ in the last 7.

Computing with sequences: Streams.

The errors of iteration formulas in floating-point arithmetic. Floating-point arithmetic is not real arithmetic, it's an approximation to real arithmetic. With IEEE double-precision arithmetic as implemented on all RISC machines (Sparc, PowerPC, MIPs, ...), but not the default arithmetic on the x86 architecture (Pentium X, X=1,2,3,4, Athlon, ...)

$$x \odot y = (x \cdot y)(1+\epsilon) \text{ and } x \oplus y = (x+y)(1+\epsilon),$$

where $|\epsilon| \leq 2^{-53}$. We can examine some coarse effects of these errors in iterations.

A well-known example is to calculate $I_n = \int_0^1 x^n e^{x-1}\,dx$, for which $0 < I_n < 1$ for all n. We might use the iteration

$$I_{n+1} = \int_0^1 x^{n+1} e^{x-1}\,dx = x^{n+1}e^{x-1}\Big|_0^1 - (n+1)\int_0^1 x^n e^{x-1}\,dx = 1 - (n+1)I_n,$$

combined with the fact that

$$I_1 = xe^{x-1}\Big|_0^1 - \int_0^1 e^{x-1}\,dx = 1 - (1 - 1/e) = 1/e.$$

So we try it:

```
(do ((n 1 (+ n 1))
     (I_n (exp -1) (- 1 (* (+ n 1) I_n))))
    ((> n 20))
  (display (list "For n = " n ", I_n = " I_n #\newline)))
```

which gives the output

```
For n = 1, I_n = .36787944117144233
```

```
For n = 2, I_n = .26424111765711533
For n = 3, I_n = .207276647028654
For n = 4, I_n = .17089341188538398
For n = 5, I_n = .14553294057308008
For n = 6, I_n = .1268023565615195
For n = 7, I_n = .11238350406936348
For n = 8, I_n = .10093196744509214
For n = 9, I_n = .09161229299417073
For n = 10, I_n = .0838770700582927
For n = 11, I_n = .07735222935878028
For n = 12, I_n = .07177324769463667
For n = 13, I_n = .06694777996972334
For n = 14, I_n = .06273108042387321
For n = 15, I_n = .059033793641901866
For n = 16, I_n = .05545930172957014
For n = 17, I_n = .05719187059730757
For n = 18, I_n = -.029453670751536265
For n = 19, I_n = 1.559619744279189
For n = 20, I_n = -30.19239488558378
```

For this iteration, any error we may have made or accumulated in I_n is multiplied by $n+1$ when computing I_{n+1}. So the relative initial error in I_1, which is bounded by

```
> (abs (exact->inexact (- (inexact->exact (exp -1)) (/ (binary-splitting-compute-e 100)))))
1.2428753672788363e-17
> (/ (log (/ 1.2428753672788363e-17 .36787944117144233)) (log 2))
-54.71640093873723
```

or well less than 2^{-53} has been multiplied by 20! or

```
> (* 1.2428753672788363e-17 (partial-factorial 0 20))
30.237939769659597
```

which is very close to the absolute value of the observed error in I_{20}:

```
(do ((n 1 (+ n 1))
     (E_n (/ (binary-splitting-compute-e 100)) (- 1 (* (+ n 1) E_n)))
     (I_n (exp -1) (- 1 (* (+ n 1) I_n))))
    ((> n 20))
  (display (list "n = " n ", E_n = " (exact->inexact E_n)
                 ", I_n = " I_n
                 ", |E_n - I_n| = "
                 (abs (- (exact->inexact E_n) I_n)) #\newline)))
n = 1, E_n = .36787944117144233, I_n = .36787944117144233, |E_n - I_n| = 0.
n = 2, E_n = .26424111765711533, I_n = .26424111765711533, |E_n - I_n| = 0.
n = 3, E_n = .20727664702865392, I_n = .207276647028654, |E_n - I_n| = 8.326672684688674e-17
n = 4, E_n = .1708934118853843, I_n = .17089341188538398, |E_n - I_n| = 3.0531133177191805e-16
n = 5, E_n = .14553294057307858, I_n = .14553294057308008, |E_n - I_n| = 1.4988010832439613e-15
n = 6, E_n = .12680235656152844, I_n = .1268023565615195, |E_n - I_n| = 8.93729534823251e-15
n = 7, E_n = .11238350406930084, I_n = .11238350406936348, |E_n - I_n| = 6.264433416447446e-14
n = 8, E_n = .10093196744559327, I_n = .10093196744509214, |E_n - I_n| = 5.0112691774018e-13
n = 9, E_n = .09161229298966059, I_n = .09161229299417073, |E_n - I_n| = 4.51014225966162e-12
n = 10, E_n = .08387707010339417, I_n = .0838770700582927, |E_n - I_n| = 4.5101464229979626e-11
n = 11, E_n = .0773522288626642, I_n = .07735222935878028, |E_n - I_n| = 4.961160787742003e-10
n = 12, E_n = .07177325364802956, I_n = .07177324769463667, |E_n - I_n| = 5.953392889779252e-9
n = 13, E_n = .0669477025756157, I_n = .06694777996972334, |E_n - I_n| = 7.739410763651922e-8
n = 14, E_n = .06273216394138015, I_n = .06273108042387321, |E_n - I_n| = 1.0835175069390246e-6
n = 15, E_n = .059017540879297774, I_n = .059033793641901866, |E_n - I_n| = 1.6252762604092308e-5
n = 16, E_n = .0557193459312356, I_n = .05545930172957014, |E_n - I_n| = 2.600442016654561e-4
n = 17, E_n = .05277111916899476, I_n = .05719187059730757, |E_n - I_n| = .004420751428312809
```

```
n = 18, E_n = .050119854958094255, I_n = -.029453670751536265, |E_n - I_n| = .07957352570963053
n = 19, E_n = .0477227557962091, I_n = 1.559619744279189, |E_n - I_n| = 1.51189698848298
n = 20, E_n = .045544884075818054, I_n = -30.19239488558378, |E_n - I_n| = 30.237939769659597
```

Here `E_n` is an approximation to the answer that uses a rational approximation with an error of 1/100! to the initial value $1/e$ of I_n. So `E_n` is not really *exact*, of course, but is is accurate to 20!/100! or about 464 bits when $n = 20$. with

So, indeed, the error in I_n is close to 20! times the error in our initial floating-point approximation of $e^{-1} = I_1$.

Note that for large n we have that $I_n \approx 1/(n+1)$, since x^n is going to be close to 0 unless $x \approx 1$, where $e^{x-1} \approx 1$. So

$$I_n = \int_0^1 x^n e^{x-1}\,dx \approx \int_0^1 x^n\,dx = \frac{1}{n+1}.$$

To see how good our rough approximation is, we have $I_{20} = .045544884075818054$, while $1/21 = .047619047619047616$.

Given (an approximation to) I_{n+1}, we can find (an approximation to) I_n with

$$I_n = \frac{1 - I_{n+1}}{n+1} \quad \text{or} \quad I_{n-1} = \frac{1 - I_n}{n}.$$

Note that this formula multiplies any error in I_{n+1} by $1/(n+1)$ to calculate I_n, i.e., it decreases any previously accumulated error before adding new errors for the floating-point subtraction and division. So we can rewrite the formula with $I_{39} \approx 1/40 = 0.025$ to get

```
> (do ((n 39 (- n 1))
     (I_n 0.025 (/ (- 1 I_n) n)))
    ((< n 1))
  (display (list "n = " n ", I_n = " I_n #\newline)))
n = 39, I_n = .025
n = 38, I_n = .024999999999999998
n = 37, I_n = .025657894736842105
n = 36, I_n = .02633357041251778
n = 35, I_n = .027046289710763394
n = 34, I_n = .027798677436835333
n = 33, I_n = .028594156545975434
n = 32, I_n = .02943654071072802
n = 31, I_n = .03033010810278975
n = 30, I_n = .03127967393216807
n = 29, I_n = .0322906775355944
n = 28, I_n = .03336928698153123
n = 27, I_n = .03452252546494532
n = 26, I_n = .035758424982779806
n = 25, I_n = .03708621442373924
n = 24, I_n = .03851655142305043
n = 23, I_n = .0400618103573729
n = 22, I_n = .04173644302794031
n = 21, I_n = .043557434407820894
n = 20, I_n = .045544884075818054
n = 19, I_n = .0477227557962091
n = 18, I_n = .050119854958094255
n = 17, I_n = .05277111916899477
```

```
n = 16, I_n = .0557193459312356
n = 15, I_n = .059017540879297774
n = 14, I_n = .06273216394138015
n = 13, I_n = .0669477025756157
n = 12, I_n = .07177325364802957
n = 11, I_n = .0773522288626642
n = 10, I_n = .08387707010339417
n = 9, I_n = .09161229298966059
n = 8, I_n = .10093196744559327
n = 7, I_n = .11238350406930084
n = 6, I_n = .12680235656152844
n = 5, I_n = .1455329405730786
n = 4, I_n = .17089341188538426
n = 3, I_n = .20727664702865395
n = 2, I_n = .26424111765711533
n = 1, I_n = .36787944117144233
```

So you see that by the time you get to $n = 20$, any initial error in I_{39}, together with any error added by subsequent floating-point operations, has been damped so much that you get a correctly rounded result for I_{20}. Even if you use 0 as an initial guess for I_{39}, the result of I_{20} is perfect with this iteration:

```
> (do ((n 39 (- n 1))
     (I_n 0. (/ (- 1 I_n) n)))
    ((< n 1))
  (display (list "n = " n ", I_n = " I_n #\newline)))
n = 39, I_n = 0.
n = 38, I_n = .02564102564102564
n = 37, I_n = .02564102564102564
n = 36, I_n = .026334026334026334
n = 35, I_n = .027046277046277045
n = 34, I_n = .0277986777986778
n = 33, I_n = .028594156535333006
n = 32, I_n = .029436540711050514
n = 31, I_n = .03033010810277967
n = 30, I_n = .0312796739321684
n = 29, I_n = .032290677535594385
n = 28, I_n = .03336928698153123
n = 27, I_n = .03452252546494532
n = 26, I_n = .035758424982779806
n = 25, I_n = .03708621442373924
n = 24, I_n = .03851655142305043
n = 23, I_n = .0400618103573729
n = 22, I_n = .04173644302794031
n = 21, I_n = .043557434407820894
n = 20, I_n = .045544884075818054
n = 19, I_n = .0477227557962091
n = 18, I_n = .050119854958094255
n = 17, I_n = .05277111916899477
n = 16, I_n = .0557193459312356
n = 15, I_n = .059017540879297774
n = 14, I_n = .06273216394138015
n = 13, I_n = .0669477025756157
n = 12, I_n = .07177325364802957
n = 11, I_n = .0773522288626642
n = 10, I_n = .08387707010339417
n = 9, I_n = .09161229298966059
```

```
n = 8, I_n = .10093196744559327
n = 7, I_n = .11238350406930084
n = 6, I_n = .12680235656152844
n = 5, I_n = .1455329405730786
n = 4, I_n = .17089341188538426
n = 3, I_n = .20727664702865395
n = 2, I_n = .26424111765711533
n = 1, I_n = .36787944117144233
```

Things are more subtle when you have iterations

$$a_{n+1} = a_n + b_n \quad \text{or} \quad a_{n+1} = a_n(1 + c_n) + d_n,$$

where $|b_n|$ and $|d_n|$ are significantly less than $|a_n|$ and $|c_n|$ is significantly less than one. We examine some examples of these cases here.

It often happens that if you do a lot of operations, the associated ϵ are roughly uniformly distributed in the interval $[-2^{-53}, 2^{-53}]$. (This doesn't happen *all* the time, though.) So if you have an iteration

$$a_{n+1} = a_n + b_n \tag{1}$$

and $|b_n| \lll |a_n|$, the error accumulates roughly as the sum of n uniform $[-2^{-53}, 2^{53}]$ random variables, i.e., the standard deviation of the error is proportional to $\sqrt{n}2^{-53}$.

On the other hand, if the iteration is

$$a_{n+1} = a_n(1 + c_n) + d_n, \tag{2}$$

where $b_n = a_n \cdot c_n + d_n$ with $|c_n| \lll 1$ and $|d_n| \lll |a_n|$, the error will grow roughly linearly in n if $c_n < 0$ and quadratically in n if $c_n > 0$. I don't know how to prove these two statements, but here are some examples.

For example, when calculating Fast Fourier Transforms, it is nice to find a fast way to calculate

$$s_j = \sin(2j\pi/N) \text{ and } c_j = \cos(2\pi j/N), \quad j = 0, 1, 2, \ldots, N.$$

If we let $\Delta = 2\pi/N$ then trigonometry gives us

$$\begin{aligned} s_{j+1} &= s_j \cos(\Delta) + c_j \sin(\Delta) \\ c_{j+1} &= c_j \cos(\Delta) - s_j \sin(\Delta) \end{aligned} \tag{3}$$

Note that (3) is roughly of the form (2) when s_j and c_j are $O(1)$, since $\cos(\Delta) \approx 1$ and $\sin(\Delta) \approx \Delta$. This is the formula that *Numerical Recipes* uses to compute its trigonometry tables.

Now

$$\sin^2 \Delta = \frac{1 - \cos(2\Delta)}{2} \quad \text{or} \quad \cos \Delta = 1 - 2\sin^2(\Delta/2),$$

so we can rewrite (3) as

$$\begin{aligned} s_{j+1} &= s_j(1 - 2\sin^2(\Delta/2)) + c_j \sin(\Delta) = s_j + (c_j \sin(\Delta) - s_j \times 2\sin^2(\Delta/2)) \\ c_{j+1} &= c_j(1 - 2\sin^2(\Delta/2)) - s_j \sin(\Delta) = c_j - (s_j \sin(\Delta) + c_j \times 2\sin^2(\Delta/2)). \end{aligned} \tag{4}$$

Formula (4) requires two more additions to calculate s_{j+1} and c_{j+1} from s_j and c_j, but it is in the form of (1) (at least when s_j and c_j are $O(1)$), and we shall see that it has less error than (3).

Finally, there is the so-called Goertzel Algorithm, which uses a two-term recursion for s_j and c_j separately:

$$\begin{aligned} s_{j+1} &= 2\cos(\Delta)s_j - s_{j-1} \\ c_{j+1} &= 2\cos(\Delta)c_j - c_{j-1}, \end{aligned} \tag{5}$$

which again can be verified by trigonometry. Here we use only two multiplications and two additions to compute s_{j+1} and c_{j+1}, but we are multiplying any previous error in s_j and c_j by 2, which magnifies it. If we assume that the error in s_j is $j^2\epsilon$ for some ϵ, then we have the error in s_{j+1} is about

$$2\cos(\Delta)j^2\epsilon - (j-1)^2\epsilon = ((2\cos(\Delta)-1)j^2 + 2j - 1)\epsilon \approx (j^2+2j+1)\epsilon - \Delta^2 j^2\epsilon - 2\epsilon \approx (j+1)^2\epsilon$$

as long as $\Delta^2 j^2 \approx 1$, which it will be if we compute for $0 \le j \le N$ and $\Delta = 2\pi/N$. So the error in s_{j+1} will be about $(j+1)^2\epsilon$. (Do not try this induction with your undergraduate students!)

So we try to compute the errors of each formula (3), (4), and (5) as accurately as possible. First, we calculate an approximation to $2\pi/N$ and call that approximation Δ. Then we compute $2\pi - N\Delta$ to high precision, convert that to a floating-point number and compute its sin and cosine, since

$$\sin(N\Delta) = \sin(N\Delta - 2\pi) \text{ and } \cos(N\Delta) = \cos(N\Delta - 2\pi).$$

We use this as our "exact" value for $\sin(N\Delta)$ and $\cos(N\Delta)$. (Note that this does not change the expected value for $\cos(N\Delta)$ since the slope of cos at $N\Delta$ is basically 0, but it does affect the correct value of $\sin(N\Delta)$.) We then compute the iterations defined by (3), (4), and (5) and compute the errors for at $N\Delta$ for $N = 1, 10, 100, \ldots 10{,}000{,}000$. Because the operations in the loop are very simple (small-integer and floating-point arithmetic) we compile the program and add some declarations for speed:

```
(declare (standard-bindings)(extended-bindings)(block)(not safe))
(define-macro (FIX . body)
  `(let ()
     (declare (fixnum))
     ,@body))
(define-macro (FLOAT . body)
  `(let ()
     (declare (flonum))
     ,@body))
(define (fixed.+ x y)
  (+ x y))
(define (fixed.- x y)
  (- x y))
(define (fixed.* x y)
  (quotient (* x y) beta^k))
(define (fixed.square x)
  (fixed.* x x))
(define (fixed./ x y)
  (quotient (* x beta^k) y))
```

```
(define (fixed.sqrt x)
  (##exact-int.sqrt (* x beta^k)))
(define (number->fixed x)
  (round (* x beta^k)))
(define (fixed->number x)
  (/ x beta^k))
(define (pi-brent-salamin)
  (let ((one (number->fixed 1)))
    (let loop ((a one)
               (b (fixed.sqrt (quotient one 2)))
               (t (quotient one 4))
               (x 1))
      (if (= a b)
          (fixed./ (fixed.square a) t)
          (let ((new-a (quotient (fixed.+ a b) 2)))
            (loop new-a
                  (fixed.sqrt (fixed.* a b))
                  (fixed.- t (* x (fixed.square (fixed.- new-a a))))
                  (* 2 x)))))))
(define beta^k (expt 10 1000))
(define two*pi (* 2 (fixed->number (pi-brent-salamin))))
(define (square x) (* x x))
(define (multiples-of-2^53 x)
  (inexact->exact (round (* x (expt 2 53)))))
(do ((N 1 (* N 10)))
    ((> N 10000000))
  (let* ((Delta (exact->inexact (/ two*pi N)))
         (N*Delta-2*pi (exact->inexact (- (* N (inexact->exact Delta))
                                          two*pi)))
         (correct-sin (sin N*Delta-2*pi))
         (correct-cos (cos N*Delta-2*pi))
         (sin-Delta (sin Delta))                  ; for formula (3)
         (cos-Delta (cos Delta))
         (two*sin^2-Delta/2                       ; for formula (4)
          (* 2 (square (sin (/ Delta 2)))))
         (two*cos-Delta (* 2 cos-Delta)))         ; for formula (5)
    (let loop ((j 0)
               (s_j-3 0.0)    ; from formula (3)
               (c_j-3 1.0)
               (s_j-4 0.0)    ; from formula (4)
               (c_j-4 1.0)
               (s_j-5 0.0)    ; from formula (5)
               (s_j-1-5 (sin (- Delta)))
               (c_j-5 1.0)
               (c_j-1-5 (cos (- Delta))))
      (if (FIX (< j N))
          (FLOAT
           (loop (FIX (+ j 1))
                 (+ (* s_j-3 cos-Delta)
                    (* c_j-3 sin-Delta))
                 (- (* c_j-3 cos-Delta)
                    (* s_j-3 sin-Delta))
                 (+ s_j-4
                    (- (* c_j-4 sin-Delta)
                       (* s_j-4 two*sin^2-Delta/2)))
                 (- c_j-4
                    (+ (* s_j-4 sin-Delta)
```

```
                    (* c_j-4 two*sin^2-Delta/2)))
                 (- (* two*cos-Delta s_j-5)
                    s_j-1-5)
                 s_j-5
                 (- (* two*cos-Delta c_j-5)
                    c_j-1-5)
                 c_j-5))
           (begin
             (display (list
                       "N = " N
                       ;", sin = " correct-sin
                       ;", cos = " correct-cos
                       ", E-s-3 = " (multiples-of-2^53 (- correct-sin s_j-3))
                       ", E-c-3 = " (multiples-of-2^53 (- correct-cos c_j-3))
                       ", E-s-4 = " (multiples-of-2^53 (- correct-sin s_j-4))
                       ", E-c-4 = " (multiples-of-2^53 (- correct-cos c_j-4))
                       ", E-s-5 = " (multiples-of-2^53 (- correct-sin s_j-5))
                       ", E-c-5 = " (multiples-of-2^53 (- correct-cos c_j-5))
                       #\newline)))))))
  N = 1, E-s-3 = 0, E-c-3 = 0, E-s-4 = 0, E-c-4 = 0, E-s-5 = 0, E-c-5 = 0
  N = 10, E-s-3 = -2, E-c-3 = 0, E-s-4 = -2, E-c-4 = 0, E-s-5 = 2, E-c-5 = 0
  N = 100, E-s-3 = 0, E-c-3 = 0, E-s-4 = 4, E-c-4 = 4, E-s-5 = -67, E-c-5 = -12
  N = 1000, E-s-3 = 8, E-c-3 = 129, E-s-4 = 14, E-c-4 = -12, E-s-5 = -22416, E-c-5 = -90
  N = 10000, E-s-3 = -8, E-c-3 = 3770, E-s-4 = -20, E-c-4 = 6, E-s-5 = -5898748, E-c-5 = 5307
  N = 100000, E-s-3 = -5, E-c-3 = 32310, E-s-4 = -157, E-c-4 = -22, E-s-5 = -518196750, E-c-5 = 32342
  N = 1000000, E-s-3 = 187, E-c-3 = 13825, E-s-4 = -125, E-c-4 = -52, E-s-5 = -2101377384, E-c-5 = 33712137
  N = 10000000, E-s-3 = -1284, E-c-3 = 526760, E-s-4 = -152, E-c-4 = 20, E-s-5 = -815709884965, E-c-
5 = 44649467
```