

# Carey Bloodworth

---

Welcome to my home page!

In the past couple of years, I've spent a lot of time developing programs to compute insanely large number of digits of pi. Why? Because the challenge was fun. In that time, though, I've learned a lot about doing large scale multiplication. (By 'large', I mean a least a million digits on up to several thousand times more than that.)

I figured it was time for me to share what I know, so that you don't have to spend as much time learning it the hard way, like I did.

I'm giving you numerous examples of big number multiplication. I'm including FFT multiplication, NTT (Pollard, prime radix FFT, etc.), Nussbaumer convolution, and Schonhage-Strassen convolution.

The example programs are public domain. I do, however, ask that if you find any of them or this site useful, please give me credit in your program, docs, web page, etc. I think that's fair, okay?

**[New this time](#) is a slightly embarrassing page about my stupidity in missing an obvious modification to [Schonhage](#) to make it work in any base, not just binary.**

**Also, I put a short note on the download page about a bug in Jason's assembly code for his demonstration 62 bit Montgomery multiply.**

[SiteMap](#)

[Pi](#) | [Multiplication](#) | [Downloads](#) | [Related Links](#) | [What's new](#) | [Contact Me](#) | [Thanks](#)  
| [To do](#)  
[Continued fractions](#) [Binary Splitting](#)

If you have comments or suggestions, feel free to send me a note. I've got a convenient feedback form on my 'contact me' page. Plus an email address if you'd rather use your mail program.

If you link to this site, please link to the main page at:

**[www.BLOODWORTH.org](http://www.BLOODWORTH.org)**

(Yup, I was lucky enough to be able to register my last name.)

This site and its contents are copyrighted.

Most of the files are distributed under their own license, often public domain, so please check them for specifics. All code snippets directly on the page itself are public domain. All papers are copyright their respective authors unless explicitly stated otherwise.

Privacy policy:

Please note that I respect your privacy as much as I like my own.  
I don't care who you are. Simple as that. Although my web and domain host  
probably keep records about visitors, I don't see any of that.

Site last updated September 28, 2001



visitors since May 1, 2001



[Home Page](#)[SiteMap](#)[Download page](#)[Feedback form](#) [Carey Bloodworth](#)

- [Pi](#)
- [Multiplication](#)
- [Downloads](#)
- [Related Links](#)
- [What's new](#)
- [Contact Me](#)
- [Thanks](#)
- [To do](#)

# Pi

To put things very simply, I haven't worked on my pi-AGM program very much since Dominique Delande computed one billion ( $2^{30}$ ) decimals with my program.

I did quite a bit of work on it, but I never really finished the improvements.

I developed a new NTT multiplication routine that was capable of reaching to at least 256g digits.

I developed a disk based multiplication module. It wasn't tuned, but it worked and showed a lot of promise.

I reduced the disk consumed down to just 3.75 times the number of decimals computed. (This is less than half what the previous program used!)

But in spite of those improvements, I never really got done. Why?

Well, because I wasn't happy with it. The code was disgusting. The disk I/O would have been excessive. (It has always been bad, but for these ranges it would have been even worse.)

Do you have any idea how hard it is to work on a program that you wrote but dislike?

With my previous v2.x program, my goal was only to do 32 million decimals on a 486/66 faster than what David Bailey did on a Cray-2 back in 1986. It turned out the program was extensible far beyond that, but that was never really my goal.

Well, now my goal is really massive computations, and that means things need to be done a bit differently.

Both in the structure of the program and even the method used.

I'm still working on a good solution, but so far I haven't found anything I like. Which is why there isn't anything of importance on this page.

Now, about the old v2.3.1 program....

Well, if you've downloaded the source code and tried to compile it with the current GNU C compiler, you'll have certainly noticed that it doesn't compile! That's because they changed the way GNU C handles inline assembly and the new structure and restrictions aren't completely compatible with the old method.

If you really need to compile that old program, the let me know. I think I have some drop in replacement modules, but right off hand I don't know where. You shouldn't really need to compile it anyway, since it's distributed with executables.

The last distributed version is available on the [download page](#).



[Home Page](#)[SiteMap](#)[Download page](#)[Feedback form](#) [Carey Bloodworth](#)[Pi](#)[Multiplication](#)• **Downloads**[Related Links](#)[What's new](#)[Contact Me](#)[Thanks](#)[To do](#)

# Downloads

My multiplication demos are available here: [muldemos.ZIP](#) (July 16, 2001)

(This contains Karatsuba, complex-FFT, wrapper-FFT, right angle-FFT, 31 bit NTT, 64 bit NTT, multi-prime 31 bit NTT, FGT, FHT (4 styles), nussbaumer, a 'balanced data' fft, Montgomery NTT31, Montgomery NTT62, wide NTT, and Schonhage-Strassen.)

Jason's public domain Pentium optimized FPU modmul is here: [ntt586.zip](#)

Jason's x86 62 bit MontMul code is here: [Mont62.txt](#)

**(WARNING: The version I have posted here has an error of some sort in it. Jason hasn't sent me a new version, so you'll need to contact him yourself. Jason Stratos Papadopoulos jasonp-at-Glue.umd.edu )**

An simple example of binary splitting (using GNU GMP) is here. [BinSplit.ZIP](#) This also includes my "pretend" code that allows me to count operation cost without actually doing the computation.

Here is the example code for working with continued fractions. [gosper.c](#) (Uses GNU GMP)

Richard Crandall's  $3 \cdot 2^k$  convolution is here: [Con32k.ps](#)

Richard Crandall's FGT paper is here: [Confgt.ps](#)

Dr. David Bailey's non-power of two convolution is here: [Convops.ps](#)

Colin Percival's paper on FFT error rates is here: [mpaper.ps](#)

A copy of Jorg Arndt's FFT paper is here: [fxtbook.ps](#) (Feb 19,2001)

An old newsgroup posting on montgomery multiplication: [Montmul.txt](#)

A paper by Cetin Koc, Tolga Acar and Burton Kalisi Jr on Montgomery Multiplication. [j37acmon.pdf](#)

A simple program to find NTT primes: [findprim.c](#)

A simple program to find NTT roots: [findroot.c](#)

Mikko Tommila's 64 bit special prime ModMul: [Raw.h](#)

My old v2.3.1 pi program source is here: [Cbpi231s.zip](#)

My old v2.3.1 pi program executables are here: [Cbpi231b.zip](#)

My public domain (and crude!) pi program is here: [piagm15.zip](#)

My old public domain pi tutorial is here: [Pitutor.zip](#)

My old public domain pi reference is here: [Pi\\_ref.txt](#)

Felix Bileski has converted the pi\_ref file to .PDF format. [pirefpdf.zip](#)  
(The .pdf file is passworded to prevent tampering. If you use GhostScript / GhostView to read PDF (rather than Adobe's reader), you'll need the decryption module. [pdf\\_sec.ps](#) Put this module into your Ghostscript directory, replacing the existing non-working 'stub' version.)

An old paper on continued fractions. [Cf6.txt](#)



# Continued fractions

[Home](#)
[Page](#)
[SiteMap](#)
[Download  
page](#)
[Feedback  
form](#)

(A plain text version of this page and some example code is available on the [download page](#).)

This document is titled: cf6.txt

It was written by Carey Bloodworth and placed into the public domain on July 19, 1996

It was based almost entirely on an unfinished continued fraction paper written by Robert William ("Bill") Gosper. Although his techniques work, the examples he gave were a bit on the complex side, often obscuring how to even do the basic operation. I started writing this etext mostly as a means to understand what he was trying to say.

## Why use Continued Fractions?

=====

I suppose you wouldn't be satisfied with "Because they are there"?

Well, it's generally accepted that patterns are easier to detect in a continued fraction than in some arbitrary positional notation base, such as our base 10.

For example, consider:

$\text{sqrt}(2)$  is 1 (2)

'e' is 2 1 2 1 1 4 1 1 6 1 1 8 1..., which can be written as  
 $2 (1 \ 2k+2 \ 1)$  and  
 $1 \ 0 \ 1 (1 \ 2k+2 \ 1)$  and  
 $(1 \ 2k \ 1)$

$4/e$  is 1 2 8 3 (1 1 1  $k+1$  7 1  $k+1$  2) and  
 $1 \ 2 (1 \ k \ 7 \ 1 \ k \ 2 \ 1 \ 1)$

$\text{sqrt}(3)$  is 1 (1 2)

$\text{sqrt}(23)$  is 4 (1 3 1 8)

$\text{sqrt}(19)$  is 4 (2 1 3 1 2 8)

$\text{sqrt}(199)$  is 14 (9 2 1 2 2 5 4 1 1 13 1 1 4 5 2 2 1 2 9 28)

(The numbers in the parenthesis repeat for ever. When there is a variable in there, that would be the repetition number.)

All of those have a fairly obvious pattern, where as their numerical evaluation, in base 10,

doesn't.

Our base 10 is a base oriented positional notation. Continued fractions are a base independant syntax. If you calculated  $5/3$  in base 8, base 10, and base 11, you'd get three very different looking results. But with a CF, you can only get one single result,  $1\ 1\ 2$ .

If you divided  $100/2.54$ , the CF would be  $39\ 2\ 1\ 2\ 2\ 1\ 4$ , exactly. But the numerical evaluation of it in base 10 would be:

$39.(370078740157480314960629921259842519685039)$ , with the number in the parenthesis repeating for ever.

If you take two 30 digit numbers and divide them, it's likely that the output would appear to be quite random in base 10. But with CF's, it should be fairly obvious. For example, using the GNU BC arbitrary precision calculator, and the first 30 digits of 'e' and 'pi', I get:

```
-----
a=271828182845904523536028747135
b=314159265358979323846264338327
scale=1000
a/b
.86525597943226508721777478964785939799206907743106713864025003113467\
477152915150092186147965484639523315325989495951554017931516419695660\
635894519433559763502699680251523583148721861239672635183600596794156\
640532945796196968668311945693959980135926667917887436307322985930931\
257156756874997180304955295356581374376673772545272251729498023995248\
077291023275512260341767923150203012807283276473738827575745441302022\
321547215330806334139789837201115950214212094129259559596404253921544\
428367036754039564167801309097386149177751437179132333658960241206397\
678616733708815052470295518367995641129127406784165928700251306376190\
960196936273328495566886224526371225698761226034437624690691390652479\
651068775001244603621352180461094925302932972004089572428080102851449\
761762482618233323356854444471227809174709162670077391294767882226045\
060458113815488386585461229894417117418633670265102140214154479743806\
354005660672512059719024607909281529473076026540779807602890846503427\
73279546541182871285734704074935024
-----
```

But when I convert it to a continued fraction, I get an exact answer:  $0\ 1\ 6\ 2\ 2\ 1\ 2\ 6\ 8\ 2\ 1\ 1\ 4\ 3\ 1\ 1\ 66\ 2\ 1\ 1\ 2\ 4\ 2\ 7\ 46\ 10\ 2\ 1\ 7\ 1\ 27\ 3\ 9\ 2\ 1\ 5\ 1\ 2\ 5\ 1\ 1\ 2\ 10\ 1\ 2\ 2\ 1\ 7\ 2\ 9\ 2\ 1\ 35\ 1\ 2\ 23$ . A rational number always generates a finite continued fraction sequence. The same can't be said about any base notation.

Incidentally, those 1,000 digits are fairly random, and would probably pass most random number tests.

So, if CF's are so great, why don't we always use them instead of regular positional base 10? Because they take more work, are more complicated and the numbers in the internal calculations can get much larger than is convenient for hand calculations. And you've still got to know regular numbers and math to be able to work the numbers during the



calculation of a CF.

But, for all that work and effort, you do get exact results. As Bill Gosper put it "...continued fractions are not only perfectly amenable to arithmetic, they are amenable to perfect arithmetic."

What is a continued fraction?

=====

There are four kinds of continued fractions. Infinite, where the terms just go on for ever, and finite continued fraction, where there are only a fixed number of terms. And those two can be of one of two types, regular (sometimes called simple), and irregular (sometimes called complex). The regular type always has a one in the numerator. The irregular type can have other whole numbers there.

I'm sure you can imagine the difference between infinite and finite continued fractions, so I'll only show the difference between regular and irregular.

Regular infinite continued fraction:

$$\begin{array}{r} \text{sqrt}(2) = 1 + \frac{1}{\phantom{1 + \frac{1}{2 + \frac{1}{2 + \frac{1}{2 + \dots}}}}} \\ \phantom{1 + \frac{1}{2 + \frac{1}{2 + \frac{1}{2 + \dots}}}} \\ \phantom{1 + \frac{1}{2 + \frac{1}{2 + \dots}}} \\ \phantom{1 + \frac{1}{2 + \frac{1}{2 + \dots}}} \\ \phantom{1 + \frac{1}{2 + \frac{1}{2 + \dots}}} \end{array}$$

Irregular infinite continued fraction:

$$\begin{array}{r} \frac{4}{\pi} = 1 + \frac{1^2}{\phantom{1 + \frac{1^2}{3 + \frac{2^2}{5 + \frac{3^2}{7 + \dots}}}}} \\ \phantom{1 + \frac{1^2}{3 + \frac{2^2}{5 + \frac{3^2}{7 + \dots}}}} \\ \phantom{1 + \frac{1^2}{3 + \frac{2^2}{5 + \frac{3^2}{7 + \dots}}}} \\ \phantom{1 + \frac{1^2}{3 + \frac{2^2}{5 + \frac{3^2}{7 + \dots}}}} \\ \phantom{1 + \frac{1^2}{3 + \frac{2^2}{5 + \frac{3^2}{7 + \dots}}}} \end{array}$$

Regular CF's are often written as only the denominators, since the numerators are always one and there isn't any point in writing them down. We would write the square root of two's CF as: 1 2 2 2 2..., or as 1 (2), where the number(s) in the parenthesis repeat forever.

I don't know how irregular CFs are written. I suppose about any way would do, as long as

there was no chance of misunderstanding. For the purpose of this paper, I'll do it with a colon ":" between each pair of numbers. The irregular CF  $4/\pi$  formula above would be: 1 1:3 4:5 9:7...

### Conversion of a rational number into a regular continued fraction

=====

Converting a rational number into a regular CF is actually fairly simple. For example, say we have 2.54 and we would like to make a CF out of it. The first thing we do is get rid of the decimal point and make it into a rational number. The number would then become 254 / 100, which is obviously the same thing, just in a different form. It doesn't matter whether it is in lowest terms or not.

We then perform Euclid's Greatest Common Divisor algorithm on it:

254			
100	2	254/100	= 2, with a remainder of 54
54	1	100/54	= 1, with a remainder of 46
46	1	54/46	= 1, with a remainder of 8
8	5	46/8	= 5, with a remainder of 6
6	1	8/6	= 1, with a remainder of 2
2	3	6/2	= 3, with no remainder
0			

We have now discovered that the finite regular CF is: 2 1 1 5 1 3. And also that the GCD of our 254/100 is 2, since that was the last divisor.

### Conversion of a regular continued fraction into a rational number

=====

You could start at the end of the finite CF and keep building up the rational number. But, if you are dealing with a CF that is infinite or is simply coming in a term at a time and you can't wait for the end, then you can start at the beginning.

This is a little bit more difficult, but it can be done with a bit of thinking. For this example, I'll use the RCF for 2.54 above.

We take the first number and we then add the next term to it, remembering that we are actually working with fractions, rather than whole numbers. We also need to remember that the CF is a regular CF, which means that the numerators are going to be one.

2	1	2*1+0	3
-	+	-	=
1	1	1*1+0	1



I know that looks odd, but please be patient, and you'll understand where the numbers are coming from, especially that "+0".

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

$$\frac{5}{2} + \frac{1}{5} = \frac{5*5+3}{2*5+1} = \frac{28}{11}$$

$$\frac{28}{11} + \frac{1}{1} = \frac{28*1+5}{11*1+2} = \frac{33}{13}$$

$$\frac{33}{13} + \frac{1}{3} = \frac{33*3+28}{13*3+11} = \frac{127}{50}$$

And we have our answer! 127/50. Hmmmm... That doesn't look like the 254/100 that we started with... Well, it isn't. Remember, we originally had 2.54 and forced it into the rational form of 254/100. It wasn't in lowest terms, because it has a GCD of two, and could have been reduced down to 127/50. I should also point out that even though this example CF resulted in a rational of lowest terms, there is no requirement in the CF math that forces this. This was just pure luck. As you can see from some of the worked examples below, for an irregular CF, it probably won't be in lowest terms.

Anyway, to explain the method, we take our current rational estimate, we multiply it by the next term of the regular CF, and then add the previous rational number we had. We don't use the numerator of the terms of the regular CF. They are always one and don't add anything, because multiplying by '1' doesn't change anything. They can be put there, but that's just one more thing to do. They do get used with irregular CFs though. In those cases, we would multiply our previous term by it.

### Notational changes

=====

We really need a more convenient way of writing all this stuff. We need some way of 'automating' all this syntax. I'm not really sure any way is really good, but we need to pick something, so I'm going to follow the lead of how I learned it.

From now on, we are going to be working in the 'equation form' of

$$\frac{aX + b}{cX + d}$$

where, the 'X' means the continued fraction that we are working with. Using this syntax allows us to do a few tricks, like adding, subtracting, multiplying or dividing the CF by a fixed amount. We can even reciprocate the CF. Doing it this way gives us a great deal of versatility.

Instead of actually writing the 'X' all the time, we are going to abbreviate it a bit further, as a 2x2 matrix of just:

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix}$$

and you'll just have to remember what we are talking about. When working with the CF's there are a few useful matrices that you should remember.

$$\begin{array}{ll} \text{A)} & \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \quad \text{B)} & \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \quad \text{C)} & \begin{pmatrix} 2 & 0 \\ 0 & 1 \end{pmatrix} \quad \text{D)} & \begin{pmatrix} 1 & 0 \\ 0 & 2 \end{pmatrix} \end{array}$$

The first is just the 'identity' matrix. If you plug the numbers into the formula above, you'll see it's just  $(1X+0)/(0X+1)$ . The next one is the reciprocal form. You can see it's just  $(0X+1)/(1X+0)$ . Example C is two times the identity. Example D is one half of the identity.

(Just to confuse things a bit more, actually, only the very first 2x2 matrix we use will fit that equation. The rest are better thought of as composed of two terms, the new, better approximation, and the old, not as good approximation. You'll understand when you see the examples.)

### Conversion of a regular Continued Fraction into a rational number

=====

Now, let's again work with that conversion of the 2 1 1 5 1 3 CF, which you should remember, is equivalent to 2.54.

We are going to write our incoming CF along the top, in backwards right to left format. The first term will start all the way over to the right, with the next term over to the left, and so on. This does take a bit of getting used to, but unfortunately, it does make some things a bit easier to deal with.

$$\begin{array}{ccccccc} 3 & 1 & 5 & 1 & 1 & 2 & \end{array}$$

We also need to have a matrix to start with. Something to 'prime the pump', so we can have a place to start. Since we aren't wanting to do anything fancy, except just convert it



to another form, we want to use the 'identity' matrix of 1;0;0;1. We write that so the left part of it is under the first CF term.

CF		3	1	5	1	1	2	
<hr/>								
							1	0
							0	1

We then take "A" times the CF, plus the previous term, which is "B". We then take "C" times the CF, plus the previous term, which is "D".

CF		3	1	5	1	1	2	
<hr/>								
						2	1	0
						1	0	1

Notice that we have a new little matrix to work with now. 2;1;1;0. We keep repeating the operations.

CF		3	1	5	1	1	2	
<hr/>								
	127	33	28	5	3	2	1	0
	50	13	11	2	1	1	0	1

As you can see, that's the same result we got before, but it looks so much neater. Except for having to work from right to left. I suppose we could do it left to right, and reverse our matrix and so on, but this is the way I learned it and my one reference shows it this way, and I'm having enough trouble keeping everything straight while I write this stuff down.

Later I may try to go from left to right and end up rewriting all of this, in which case, you'll never know about all of this.... <BG>

You can also see that each new term is a better and better approximation to our original number, until finally we reach the last term of the finite CF and we have all the 'information' and we are able to reconstruct the original number exactly. This is what I meant when I previously said that except for the initial matrix, it was better to think of it as new and old approximations. You can also see that each approximation goes back and forth between being too high, and being too low.

### Conversion of an irregular Continued Fraction into a rational number

Alright, so we've now come up with a general way of converting a regular CF into a rational number. We can also do the same with an irregular CF. In this case, we write the numerators and denominators of it on the top, one above the other, and when we do our math to generate a new rational number, we multiply the old number by the numerator of the CF. For example, I'll use the continued fraction for pi given at the beginning.

First we need to write down the terms:

...	100	81	64	49	36	25	16	9	4	1
....	19	17	15	13	11	9	7	5	3	1

---

Next we need to chose our starting matrix. Well, that depends on what you want. If you want the rational number for  $4/\pi$ , then you could use the identity function 1;0;0;1. Frankly though, I'd rather end up with normal, regular,  $\pi$  ratios. Ratios like 22/7, 333/103, and 355/113. For that, we need to remember the 'formula' that our initial matrix represents. We need the reciprocal of all this, so our 'pi' factor will be on the bottom. And that '4' needs to be on the top, so that pi can divide into it. So, our matrix is 0;4;1;0.

...	100	81	64	49	36	25	16	9	4	1	(1)
....	19	17	15	13	11	9	7	5	3	1	

---

										0	4
										1	0

---

So, to get our next set of numbers, we multiply "A" times the denominator right above, then add "B" times the numerator right above it. (This requires setting a fake '1' as a numerator above the '4' in the initial matrix. I'm not quite sure how to get around this. It doesn't seem like 'clean' mathematics. I guess you could think of it as being the '1' numerator over the initial '1'.) We then do "C" and "D" the same way.

...	100	81	64	49	36	25	16	9	4	1	(1)
....	19	17	15	13	11	9	7	5	3	1	

---

					6976	640	76	12	4	0	4
					2220	204	24	4	1	1	0

---

Now, as you can see, these numbers are starting to get a little big. What I'm going to do is reduce them to their lowest terms. You have to do that as a group of 4, since we are dealing with a 2x2 matrix. Basically, you just find the GCD of all four numbers. In this case, we can divide by 4. And we end up with:

...	100	81	64	49	36	25	16	9	4	1	(1)
....	19	17	15	13	11	9	7	5	3	1	

---

					6976	640	76	12	4	0	4
					2220	204	24	4	1	1	0

---

					1744	160					
					555	51					

---

And we continue on:

...	100	81	64	49	36	25	16	9	4	1	(1)
....	19	17	15	13	11	9	7	5	3	1	

---



			6976	640	76	12	4	0	4
			2220	204	24	4	1	1	0
			-----						
6598656	364176	23184	1744	160					
2100420	115920	7380	555	51					

And as you can see, the numbers you are working with constantly get larger and larger.

Unfortunately, that is simply part of continued fractions and has to be accepted. You can also see that the final numbers 6598656/2100420 can be reduced down to 183296/58345 and results in 3.141588825. That's not as good as 355/113's 3.14159292. We could however, eventually reach a rational number that has at least as good of precision as 355/113.

In fact, I think I'm going to continue this example until we do.

			121		100		81		64
			21		19		17		15
			-----						
							6598656		364176
					(reduce by 36)		2100420		115920
							-----		
2189751040	86352640	3763456					183296		10116
697019400	27486900	1197945					58345		3220

Hmmm... Well. It looks like we've just jumped over the precision that 355/113 gives.

3763456/1197945	is	3.141593312
355/113	is	3.141592920
86352640/27486900	is	3.141592540
2189751040/697019400	is	3.141592673
pi	is	3.141592653

Why is this? Why didn't we get 355/113? Well, two reasons. First, it's quite possible I made a mistake in my math. (I didn't, but it is possible, especially when you are doing this by hand.) The second is that 355/113 was done from the regular continued fraction, and not the irregular continued fraction. Although you get comparable precision, the actual rational numbers you get in the process are different.

If we were to convert our irregular CF into a regular CF, and truncate that at 3 7 15 1, and then convert that into a rational number, we would get 355/113. Just as proof, I'll show you:

2189751040	
697019400	3
98692840	7
6169520	15
6150040	1

19480 And this is as far as we need to go.

(Notice that there is still information left that we could extract to get an even better approximation. But we don't want that good of precision right now, so there is no point in doing it.)

Then we take 3 7 15 1, and convert it to a rational number and:

CF		1	15	7	3	
	355	333	22	3	1	0
	113	106	7	1	0	1

And, as I'm sure you noticed, those are all the common approximations that everybody knows.  $3/1$ ,  $22/7$ ,  $333/106$ , and  $355/113$ .

I should also point out that they alternate between being a little low and a little high.

$$\begin{aligned}
 3/1 &= 3.0000000 \\
 22/7 &= 3.1428571 \\
 333/106 &= 3.1415094 \\
 355/113 &= 3.1415929 \\
 \pi &= 3.14159265\dots
 \end{aligned}$$

So each RCF term we input while creating our matrix will cause our answer to go back and forth between a little low to a little high, with the error becoming less and less. So if we want to know how much of our answer is correct, we have to use two terms and then only accept the digits that are the same. For example, by using  $333/106$  and  $355/113$ , we can know for certain that  $\pi=3.1415$  is correct to 4 decimals.

If we had wanted to, we could have continued the rational to RCF conversion above. We would have gotten:

```

2189751040
 697019400   3
  98692840   7
   6169520  15
    6150040   1
     19480 315
      13840   1
       5640   2
        2560   2
         520   4
          480   1
           40  12

```

That would give us 3 7 15 1 315 1 2 2 4 1 12. And just for the record,  $\pi$  has a RCF of 3 7 15 1 292 1 1 2... So, previously, if we had continued, it wouldn't have done us much



good, because you can already see that the next term is wrong. It's close, but still wrong. That just happens to be the best guess with the amount of information it had been given. More information gets you a better answer. When I stopped at  $3\ 7\ 15\ 1$ , that was because I already knew that it would work out to  $355/113$ .

Just as a comparison to our  $3\ 7\ 15\ 1\ 315\ 1\ 2\ 2\ 4\ 1\ 12$ , if we had converted the previous rational, we would have gotten:  $3\ 7\ 15\ 1\ 205\ 4\ 1\ 11$ , and we still would have been able to get  $355/113$ . Also, you can see, that the term that should be 292 has a bound of 315 and 205.

Of course, it's a little awkward to convert a whole string of ICFs into some enormous rational and then convert that into a RCF. But we don't have to wait. We can do it on the fly.

### Conversion of an Irregular CF to a Regular CF.

=====

Conversion of an ICF to a RCF is actually very similar to converting it to a rational number. The difference is that every so often, the matrix can 'spit out' a term of the RCF. It's some what like reduction, except it doesn't have to divide evenly, it's just that  $A/C$  and  $B/D$  have to have the same integer result when you divide. This is because  $A/C$  and  $B/D$  represent the new and old approximations, and they also represent the high and low estimates of the number, so if the high and low bounds agree about an output term, then that term has been 'solved'.

I'll show you. I'll write the output terms along the far right side, since I have to put them somewhere.

...	100	81	64	49	36	25	16	9	4	1	(1)
....	19	17	15	13	11	9	7	5	3	1	

---

							76	12	4	0	4
							24	4	1	1	0

Notice that  $76/24=3.16$  and  $12/4=3.0$  and that both have the same integer.... So we can spit out a '3' and create our new matrix. We do that by bringing down the remainder.

...	100	81	64	49	36	25	16	9	4	1	(1)
....	19	17	15	13	11	9	7	5	3	1	

---

							76	12	4	0	4
							24	4	1	1	0
							4	0			

Notice that the new matrix  $24;4;4;0$  has a GCD of 4, so we can reduce it if we want. I'm going to do that simply to keep the numbers small, but you of course don't have to.

(Note: Even though there is a '0' in the matrix, 4 is still the GCD because you can divide 4 into 0 with out any remainder.)

I'd also like to point out that the matrix  $24;4;4;0$  means that the next output is somewhere between  $24/4=6$  and  $4/0=\text{infinity}$ . So we do need to consume more terms to make those bounds a wee bit closer together.

$$\begin{array}{cccccccccccc}
 \dots 100 & 81 & 64 & 49 & 36 & 25 & 16 & 9 & 4 & 1 & (1) \\
 \dots 19 & 17 & 15 & 13 & 11 & 9 & 7 & 5 & 3 & 1 & \\
 \hline
 & & & & & & 76 & 12 & 4 & 0 & 4 \\
 & & & & & & 24 & 4 & 1 & 1 & 0 & 3 \\
 & & & & & (reduce \times 4) & 4 & 0 & & & & \\
 & & & & & & \hline
 & & & & 555 & 51 & 6 & 1 & & & & \\
 & & & & 79 & 7 & 1 & 0 & & & & 
 \end{array}$$

And we can spit out another term of the regular CF because  $51/7=7.2$  and  $555/79$  is  $7.02$ .

$$\begin{array}{cccccccccccc}
 \dots 100 & 81 & 64 & 49 & 36 & 25 & 16 & 9 & 4 & 1 & (1) \\
 \dots 19 & 17 & 15 & 13 & 11 & 9 & 7 & 5 & 3 & 1 & \\
 \hline
 & & & & & & 76 & 12 & 4 & 0 & 4 \\
 & & & & & & 24 & 4 & 1 & 1 & 0 & 3 \\
 & & & & & (reduce \times 4) & 4 & 0 & & & & \\
 & & & & & & \hline
 & & & & & & 555 & 51 & 6 & 1 & & 7 \\
 & & & & & & 79 & 7 & 1 & 0 & & \\
 & 16416 & 1044 & & & & & & & & & \\
 & 1088 & 72 & 2 & 2 & & & & & & & 
 \end{array}$$

And as you can see, you just keep consuming terms and outputting digits whenever you are able.

The larger the output terms, the more information inputted is required to generate them. That 7 came out after processing several terms. The next output would be a 15 and we need a bit more information consumed to be able to know exactly what we can output.

### Conversion from regular continued fraction to decimal

=====

You've got several ways to do this. If it's a finite continued fraction, you can simply start at the bottom and actually evaluate it. Or you could convert it to a rational number and then evaluate that. If it's an infinite continued fraction, or a long finite continued fraction, you can't afford to wait until the end. It has to be done 'on the fly', while the terms are still coming in.

It's actually very much like converting from an irregular CF to a regular CF, (or simply performing the identity matrix  $1;0;0;1$  to a regular CF) except that after you spit out a term, you multiply your new matrix numerator by 10.



I'll show you by using the regular CF for pi: 3 7 15 1 292... It would work similarly with irregular CF's except you have to multiply by the numerator, just like before.

	1	15	7	3	
-----					
		22	3	1	0
		7	1	0	1

Notice that 22/7 and 3/1 both have an integer part of 3, so we can spit out a 3. To create our next matrix, put 10 times the remainders in the top part, and we bring down the bottom part. And then we start consuming more input terms.

		1	15	7	3	
-----						
			22	3	1	0
			7	1	0	1
						3
			-----			
	150	10	0			
	106	7	1			1
	440	30				
	106	7				4
	180	160	20			
	113	106	7			1
	670	540				
	113	106				

Note that it is possible to spit out more than one decimal digit for each input consumed. This is because the inputs can be larger than our base 10 and may contain enough information for more than one of our digits.

You can also output more than a single digit at a time. If you'd like, you can multiply by 100 instead of 10 and get two digits at a time. Of course, you'll have to consume more terms to have enough information in the matrix to be able to spit out two digits at a time, so there isn't really anything to be gained by doing it this way.

It is also possible to delay the output and continue consuming input terms and then spit out a burst of output digits.

And it is possible to do this output with an irregular continued fraction, just like we are doing with a regular one. You just have to multiply the old term by the numerator, just like we were doing before when we were converting a ICF to a RCF, or when we were converting a ICF to a rational number.

Addition, Subtraction, Multiplication, Division of a CF by a number

=====

Not too hard. Remember our formula?

$$\frac{aX + b}{cX + d}$$

Let's take our original 2.54 CF of 2 1 1 5 1 3.

Let's add three to it. Our matrix will be 1;3;0;1

CF		3	1	5	1	1	2	
		277	72	61	11	6	5	1
		50	13	11	2	1	1	0

And we get 277/50, which is 5.54.

Let's multiply it by 8, then add 1, and then divide by 2! Our Matrix will be 8;1;0;2

CF		3	1	5	1	1	2	
		1066	277	235	42	25	17	8
		100	26	22	4	2	2	0

And we get 1066/100, which is 10.66. And  $((2.54*8)+1)/2$  is 10.66.

See how easy all that is?

Addition, Subtraction, Multiplication, Division of a CF by a CF

=====

Doing the basic four basic functions with two continued fractions is, unfortunately, a bit more difficult.

Again, I'm going to use the CF for 2.54, 2 1 1 5 1 3. I'm also going to use the CF for 3.72, 3 1 2 1 1 3.

So, we have two CFs:

$$x = x_1 x_2 x_3 \dots$$

$$y = y_1 y_2 y_3 \dots$$

Our old:



$$\frac{aX + b}{cX + d}$$

could do all sorts of things to one CF, but it simply won't work for two CFs.

Instead, we need to use something like:

$$\frac{aXY + bX + cY + d}{eXY + fX + gY + h}$$

to be able to cover all the possibilities of what can be done.

And obviously, a 2x2 matrix isn't going to be able to handle those eight numbers, so we are going to arrange things as a 2x2 matrix, with a second matrix 'floating' underneath the first.

It'll look like this:

$$\begin{array}{cc} b & d \\ f & h \\ a & c \\ e & g \end{array}$$

Then the four basic functions all come down to a starting matrix of:

$$\begin{array}{cc} x+y = & \begin{matrix} 1 & 0 \\ 0 & 1 \end{matrix} & x-y = & \begin{matrix} 1 & 0 \\ 0 & -1 \end{matrix} \\ & \begin{matrix} 0 & 1 \\ 0 & 0 \end{matrix} & & \begin{matrix} 0 & 1 \\ 0 & 0 \end{matrix} \end{array}$$

$$\begin{array}{cc} x*y = & \begin{matrix} 0 & 0 \\ 0 & 1 \end{matrix} & x/y = & \begin{matrix} 1 & 0 \\ 0 & 0 \end{matrix} \\ & \begin{matrix} 1 & 0 \\ 0 & 0 \end{matrix} & & \begin{matrix} 0 & 0 \\ 0 & 1 \end{matrix} \end{array}$$

I know this isn't quite as simple looking as before, but that's how it has to be done. It's not quite as random and arbitrary as it appears though.

If you'll notice, the left half contains all of the X terms and the bottom contains all of the Y terms. The bottom left corner contains the term that has both X and Y. And the upper right corner contains the integer. The first matrix is the numerators. The second matrix that floats under the first is the denominators.

We write our 'x' CF along the top, and the 'y' along the right. (I know, I really need to rework all this stuff for left to right conventions!)

And assuming we want to add our two CFs, we'd use the 1;0;0;1;0;1;0;0 matrix.

[illegible]

Okay, notice that E and G are zeros. Since E is on the left, it is effected only by the X term, and since G is on the right, it is effected only by the Y term. We have to input terms from both CF's before we can do much. I'll input terms from Y, going downwards, until both of its denominators are non-zero, then I'll do the same by going left for the X. The math is fairly simple. We take 2x2 matrix, either on the bottom or the left depending on which direction we are going, multiply it by the input term, and then add it to the previous 2x2 matrix.

[illegible]

After consuming two terms of  $Y$ , both denominators on the right are non-zero. We now have to get rid of the zero denominators on the left.

3      1      5      1      1      2



$$\begin{array}{cccc|c}
 & & & & 1 & 0 & & \\
 & & & & 0 & 1 & & \\
 & & & & 0 & 1 & 3 & \\
 & & & & 0 & 0 & & \\
 6 & 5 & 1 & 3 & 1 & & 1 & \\
 1 & 1 & 0 & 1 & & & & \\
 7 & 6 & 1 & 4 & & & 2 & \\
 1 & 1 & 0 & 1 & & & & \\
 & & & & & & 1 & \\
 & & & & & & 1 & \\
 & & & & & & 3 & 
 \end{array}$$

Okay, our 2x2x2 matrix now has non-zero denominators. Now we start inputting more terms. Which one depends on whether the integer part of the bottom (7/1 & 6/1) is equal, and whether the integer part of the left (6/1 & 7/1) is equal. If the bottom differs, then going left will help. If the left half differs, then going down will help. If none of them match, then it doesn't really matter. Since I was going left before, I'll keep going left.

$$\begin{array}{cccccc|c}
 3 & 1 & 5 & 1 & 1 & 2 & & \\
 \hline
 & & & & & & 1 & 0 & \\
 & & & & & & 0 & 1 & \\
 & & & & & & 0 & 1 & 3 \\
 & & & & & & 0 & 0 & \\
 61 & 11 & 6 & 5 & 1 & 3 & 1 & \\
 11 & 2 & 1 & 1 & 0 & 1 & & \\
 72 & 13 & 7 & 6 & 1 & 4 & 2 & \\
 11 & 2 & 1 & 1 & 0 & 1 & & \\
 & & & & & & 1 & \\
 & & & & & & 1 & \\
 & & & & & & 3 & 
 \end{array}$$

The bottom is now the same (72/11=6 & 13/2=6), so we stop going left and start going down.

$$\begin{array}{cccccc|c}
 3 & 1 & 5 & 1 & 1 & 2 & & \\
 \hline
 & & & & & & 1 & 0 & \\
 & & & & & & 0 & 1 & \\
 & & & & & & 0 & 1 & 3 \\
 & & & & & & 0 & 0 & \\
 61 & 11 & 6 & 5 & 1 & 3 & 1 & \\
 11 & 2 & 1 & 1 & 0 & 1 & & \\
 72 & 13 & 7 & 6 & 1 & 4 & 2 & \\
 11 & 2 & 1 & 1 & 0 & 1 & & \\
 205 & 37 & & & & & 1 & \\
 33 & 6 & & & & & & 
 \end{array}$$

1

3

Okay,  $72/11=13/2=205/33=37/6=6$ , so we can now output a term. Since I have to put them somewhere, I'll put them at the far left. To spit out a term, we subtract six times the denominator matrix  $(11;2;33;6)$  from the numerator matrix  $(72;13;205;37)$  to get the new denominator matrix. The old denominator matrix becomes our new numerator matrix.

	3	1	5	1	1	2			
	-----						1	0	
							0	1	
							0	1	3
							0	0	
	61	11	6	5	1	3			1
	11	2	1	1	0	1			
	72	13	7	6	1	4			2
	11	2	1	1	0	1			
	205	37							
	33	6							
	-----								
6	11	2							
	6	1							
	33	6							1
	7	1							
									1
									3

And then check to see if we can output a second term, which might happen since we have new denominators. In this case, we can't so we continue consuming terms. Since both the bottom and the left have different integer quotients, it doesn't matter which direction, so I'll keep heading down.

	3	1	5	1	1	2			
	-----						1	0	
							0	1	
							0	1	3
							0	0	
	61	11	6	5	1	3			1
	11	2	1	1	0	1			
	72	13	7	6	1	4			2
	11	2	1	1	0	1			
	205	37							
	33	6							
	-----								
6	11	2							
	6	1							



33	6	1
7	1	
44	8	1
13	2	
77	14	3
20	3	

The left side has finally gotten the same integer  $44/13=77/20$ , so I now start moving left. (Good thing too, I only had one more term left for Y. It would have been a rather bad example if I consumed all the input terms and still couldn't do anything!)

	3	1	5	1	1	2		
							1	0
							0	1
							0	1
							0	0
	61	11	6	5	1	3	1	
	11	2	1	1	0	1		
	72	13	7	6	1	4	2	
	11	2	1	1	0	1		
	205	37						
	33	6						
6		11	2					
		6	1					
		33	6				1	
		7	1					
	52	44	8				1	
	15	13	2					
	91	77	14				3	
	23	20	3					

We can now output a 3, since  $52/15=44/13=91/23=77/20=3$ .

	3	1	5	1	1	2		
							1	0
							0	1
							0	1
							0	0
	61	11	6	5	1	3	1	
	11	2	1	1	0	1		
	72	13	7	6	1	4	2	
	11	2	1	1	0	1		
	205	37						
	33	6						
6		11	2					

			6	1	
			33	6	1
			7	1	
		52	44	8	1
		15	13	2	
		91	77	14	
		23	20	3	
		-----			
3		15	13		
		7	5		
		23	20		3
		22	17		
		84	73		
		73	56		
		-----			
1	83	22	17		
	6	1	3		
	275	73	56		
	50	11	17		

At this point, we have consumed all of the input and have reached our final matrix. But, we still can output some terms. The AE term, at bottom left, contains our final ratio for the rest of the computation. And because we've consumed all of our input, that ratio is exact. So, if we do a Euclid's on it, we get

```

275
50  5
25  2
0

```

And 5 and 2 are the last two terms of our computation. Let's do a check. Our resulting CF was 6 3 1 5 2.

		2	5	1	3	6	
		-----					
	313	144	25	19	6	1	0
	50	23	4	3	1	0	1

And  $313/50 = 6.26$ . And our original desire was  $2.54 + 3.72$  and that does equal 6.26!

The other operations, subtraction, division, and multiplication are all done the same way, except with a different starting  $2 \times 2$  matrix. It is also possible to do this with irregular continued fractions. You just do it similarly to the way you did it when you were working with only one CF.

I think the most obvious thing that you've probably noticed is how much effort is required to do the four basic math operations with two continued fractions! It is also very error



prone, and I had to rework my example 4 complete times before it finally came out right!

## How to perform square roots on numbers

=====

Let's calculate the square root of two. That's just a simple rational number, but it will help show the math. We'll work with a full CF later.

Okay, now let's say that  $X$  is the CF of our 2. And let's say  $Y$  is the square root of it,  $Y = \sqrt{X}$ . Let's rewrite that as  $Y = X/Y$ .

Our operations will actually consume its own output. It will do this by working such as:

$$\begin{array}{rcccl} & & x & & \\ ax+b & a & b & & \\ cx-a & c & -a & x & \\ b+2ax-cxx & a-cx & & & \end{array}$$

Got it? Right! We will again use our 2x2 matrix. Since we are wanting the  $Y$  in  $Y = X/Y$ , and our own number we are using is just two, we will use the matrix  $0;2;1;0$ . The two is our two, and the one is the  $Y$  in the denominator.

$$Y = \frac{0Y+2}{1Y+0}$$

Feedback:

$$\begin{array}{cc} 0 & 2 \\ 1 & 0 \end{array}$$

We now have to guess what our first term will be. Let's say it's 5. Putting that into our  $Y$  formula gives us  $2/5$ , with an integer part of 0. Nope. 5 and 0 aren't the same. Let's try 3. We get  $2/3$ . Nope. Try 1.  $2/1$ . No, but it can't be less than 1, and it has to be somewhere between 1 and 2 because we got two when we used 1. (There is probably a better way to do this, but I don't know what, and my paper that I'm going by doesn't say.)

Feedback:

$$\begin{array}{cccc} & & 1 & \\ & & 0 & 2 \\ 1 & 1 & 0 & 1 \\ 1 & -1 & 2 & \end{array}$$

Our next term is done the same way. We make a guess, put it into our little  $Y$  equation. We can average  $Y$  and  $F(Y)$  to get a much closer answer. If  $x = F(Y)$  and  $Y = F(x)$ , then we chose the smaller of the two numbers. In our case,  $F(2)=3$  and  $F(3)=2$ . So we pick 2. Then we output that two, then immediately input our output.

Feedback:

$$\begin{array}{ccccccc}
 & & & 2 & & 1 & \\
 & & & & & & \\
 & & & & & 0 & 2 \\
 & & 1 & & 1 & 0 & 1 \\
 1 & & 1 & & -1 & 2 & 2 \\
 1 & & -1 & & 3 & & 
 \end{array}$$

Notice that our 1;1;1;-1 matrix is the same matrix we had before! Since our only input is our output, the matrix is entirely dependant upon itself. In otherwords, since our matrix is repeating, our output will repeat, and we've reached a pattern that we can quit at. The Cf of the square root of two is 1 (2), where the number in the parenthesis repeats for ever.

If all of that sounds a bit confusing and a lot like trial and error, well, I feel the same way. I have however found a second method in an article by Robert T. Kurosaka that is much more workable than the method Bill Gosper shows.

Set up a chart like:

P 0  
Q 1  
R

The 'R' will be our square root result. I don't know where the 0 and 1 come from. My guess is it's somewhat akin to Gosper's matrix. The first 'R' will just be the integer square root of the number we are wanting. It's not too hard to know that the integer part of the square root of two is one. So, we end up with:

P 0  
Q 1  
R 1

We then calculate the next P. This is the previous R times the previous Q minus the previous P. So, we do  $1*1-0$  and get '1'.

P 0 1  
Q 1  
R 1

We next calculate the Q. This is the original number minus the square of the current P, then all divided by the previous Q. So we do:  $(2-(1*1))/1$  and get 1.

P 0 1  
Q 1 1  
R 1

We then calculate the R. We take the first R, add the current P, then divide by the current Q, then take the integer part. So, we do  $\text{INT}((1+1)/1)$  and get 2.



P 0 1  
 Q 1 1  
 R 1 2

We keep doing this until the R we just calculate is twice what the first R is. Since our first R is 1, and we just calculated 2, we can stop, knowing that the CF for the square root of two is  $1(2)$ , with the number in the parenthesis repeats for ever.

According to Kurosaka, if our number is  $N^2+1$  (one more than a square, such as 17), it will have a CF of  $n(2n)$ . If it is  $N^2-1$ , it will have a CF of  $n-1(1\ 2n-2)$

### Conversion of a rational number into an irregular continued fraction

---

It's also possible to convert it into an irregular continued fraction.

For example, the 2.54 would be organized as:

$$\begin{array}{rcl}
 2.54 = & 2 + & \frac{1}{\text{-----}} \\
 & & 10 \\
 & 0 + & \frac{\text{-----}}{\text{-----}} \\
 & & 1 \\
 & 5 + & \frac{\text{-----}}{\text{-----}} \\
 & & 10 \\
 & 0 + & \frac{\text{-----}}{4}
 \end{array}$$

Notice the patter of alternating 1's and 10's in the numerator, and the 0's that alternate with the digits of the original number? Also notice that even though

You end up with a matrix alternating with our estimate and a 1:0.

This method isn't really practical, because, frankly, it's a lot easier to just convert it to a regular CF like we did in the very beginning. But it does work, and you should know about it.

### Conversion of a series into a continued fraction

---

You use the same basic idea as converting a rational into an irregular continued fraction, except instead of our numerators being 10, you use whatever denominator. And for the denominator, you use whatever numerators yours are.

For example, let our series be the classic Gregory arctangent series for  $\arctan(1)$ .  $4/1 - 4/3 + 4/5 - 4/7 + 4/9 - 4/11 \dots$ . Incidentally, this will eventually calculate pi, but the convergence is so slow that is not even close to being practical. But it does make a good example because all the numbers are small, and we have alternating signs, so you can see how to handle that.

If we were to evaluate this by hand, we would get  $4/1$ ,  $8/3$ ,  $44/15$ ,  $304/105$ ,  $2740/945$  and  $30136/10395$ .

			11	1	9	1	7	1	5	1	3	1	1
			0	4	0	-4	0	4	0	-4	0	4	
-----													
30136	1	2740	1	304	1	44	1	8	1	4	1	0	
10395	0	945	0	105	0	15	0	3	0	1	0	1	

So, as you can see, we do indeed get the right rationals, alternating with  $1/0$ .

BUT, this method has a serious drawback. You can't extract the regular CF during the conversion. You have to wait until you are completely done, and then do it.

I don't know how to get around this very serious problem. My reference doesn't say, I can't find any others, and I can't figure it out!

In short, you are just going to have to without the ability to convert an infinite series into a CF.

How to perform square roots on continued fractions

=====

Detecting patterns in continued fraction math

=====

Conversion of a infinite series into an irregular continued fraction

=====

Don't know. Probably some what similar to the conversion of a decimal number.

Comparisons, Rounding, Truncation, and approximations of CFs

=====

A few things you should know about CF terms

=====



A few CF sequences you should know

=====

The addition of, or removal of, an initial 0 to the CF stream simply inverts the entire result.

The sequence  $a\ 0\ b$  is equivalent to the single term  $a+b$ .

The zero term 'rule' lets you add a second zero in front of one already there, or delete the zero, and still get the same result.

A sequence such as  $\dots 1\ 2\ 3\ 4\ 5\ 0\ -5\ -4\ -4\ -3\ -2\ -1\dots$  is actually the same as if you didn't input those terms at all. Because there is a zero in the middle and you are 'subtracting' the terms you had just put in, they erase each other. This lets continued fraction work be reversible, where you can input or output a zero and then input or output the reverse sequence of numbers with the opposite sign.

This condition can be masked by the rule:

$$-a\ -b\ -c\ -d\ \dots = -a-1\ 1\ b-1\ c\ d$$

Hurowitz numbers are CFs that can be written in a parenthesis notation using polynomials in  $k$ . For example:

$$\begin{aligned} e &= 2\ (1\ 2k+2\ 1) \\ e &= 1\ 0\ 1\ (1\ 2k+2\ 1) \\ e &= (1\ 2k\ 1) \end{aligned}$$

And because of the 'zero rule', they are all equivalent.

Square roots of rationals are in the form:

$$a\ (b\ c\ \dots\ c\ b\ 2a) = (a\ b\ c\ \dots\ c\ b\ a\ 0)$$

And another nonsense sequence you need to watch out for is:

$$\dots -1\ 1\ -1\ 1\ -1\ 1\ \dots$$

When detected, you can shut off output until they stop. You can also discard three pairs at a time, since the only effect is to negate the entire state matrix.

[Home Page](#)[SiteMap](#)[Download page](#)[Feedback form](#) [Carey Bloodworth](#)[Pi](#)• **Multiplication**[Downloads](#)[Related Links](#)[What's new](#)[Contact Me](#)[Thanks](#)[To do](#)

# Multiplication

[Schoolboy](#) | [Karatsuba](#) | [Transforms](#) | [Big Digit](#) | [Odd sizes](#) | [Performance](#)

I'm giving you numerous examples of big number multiplication. The example programs are public domain. I do, however, ask that if you find any of them or this site useful, please give me credit in your program, docs, web page, etc.

Multiplication is surprisingly difficult to do fast.

The [normal method](#), the kind you learned in school, is a  $N^2$  algorithm. That means every time you double the length of the number, you have to do 4 times as many operations. That gets rather large very quickly.

There are actually several methods that can improve that. Each has their own strength and weaknesses. And naturally, the faster ones are more complicated.

The first method is called [Karatsuba](#), after the the author. It's also often called the 'Factal Mul' because if you see a picture of the way it recursively breaks the data, it looks like a fractal. Others call it "Divide & Conquer". This method has  $O(N^{1.585})$  growth.

The next method is based [Fast Fourier Transforms](#). There are actually several kinds, but they all share the same basic ideas. These methods have  $O(N \cdot \log_2(N))$  growth.

One method that uses FFTs are the '[big digit](#)' variety. This is where you treat a group of digits as a single 'digit' and do, well, basically a schoolboy style.

Another method is related to the FFT method, and those are what I call 'fake transform' or '[symbolic transform](#)' methods. Meaning you don't actually do any multiplications to do the transform. This includes [Nussbaumer](#) and [Schönhage-Strassen](#) methods. These have a growth of about  $O(N \cdot \log_2(N) \cdot \log_2(\log_2(N)))$  which is slightly less than a regular FFT. But these methods tend to have more overhead (the 'O' part of the cost formula) and actually usually run slower.

Most big number multiplications tend to work with power of two lengths. That's due to the transforms working more efficiently with



those sizes. In the real world you may need to multiply other sizes.  
That's discussed [here](#).

[Home Page](#)[SiteMap](#)[Download page](#)[Feedback form](#)[▲ Multiplication](#)

- **Schoolboy**

[Karatsuba](#)[Transforms](#)[Big Digit](#)[Odd sizes](#)[Performance](#)

# Schoolboy

The 'school boy' method should be pretty familiar. It's the same old stuff you've been doing since you were a kid.

```

      4711
x   6397
=====
              7
             7
          49
         28
        9
       9
      63
     36
    3
   3
  21
 12
  6
 6
42
24
=====
30136267

```

It requires  $O(N^2)$  operations. It takes  $N*N$  operations. That means everytime you double the length, it takes 4 times as many operations.

It's the simplest multiplication method known. It's also the slowest, although under certain conditions, namely short lengths, it may actually run faster because it has less overhead.

Enough said.



[Home Page](#)[SiteMap](#)[Download page](#)[Feedback form](#) [Multiplication](#)[Schoolboy](#)• [Karatsuba](#)[Transforms](#)[Big Digit](#)[Odd sizes](#)[Performance](#)

# Karatsuba

In the early 60's, it became known that you could 'divide and conquer' the multiplication. This discovery is attributed to several people, and is called several things, including 'Fractal mul', 'Karatsuba mul' and 'divide & conquer mul'.

It's doubtful that anybody knows for certain who developed this method. It's generally accredited to A. Karatsuba, but several references, including D. Knuth and D. J. Bernstein suggest that what he presented in 1962 was somewhat different and more complicated, and that he didn't even develop it alone. That uncertainty is why I myself usually call it the "fractal mul" method.

There are several related methods. I'm going to show you the one that Knuth describes, since it's the most widely available. (This isn't what Karatsuba himself described. Yet another reason to actually call this method "fractal" mul or "divide and conquer" mul.)

The method is fairly simple. If you have 'A' and 'B' you divide them into left and right halves (A1 & A2 and B1 & B2, with R being the size of the halves) and you use the formula

$$a * b \text{ is: } A2B2(R^2+R) + (A2-A1)(B1-B2)R + A1B1(R+1)$$

The multiplications by 'R' is just shifting (or adding on the bias), since we deliberately chose it to be half our width of our numbers.

If we are squaring a number, we can do it a little faster if we use:

$$A^2 \text{ is: } A2^2(R^2+R) - ((A2-A1)^2)R + A1^2(R+1)$$

We then recursively call these formulas until the chunks are small enough to use some other faster or more convenient method. (Often the schoolboy method will be faster for small sizes, because it has less overhead.)

Since we have to halve the numbers (into 'left' and 'right' parts) the algorithm works best when the length of the numbers are the same size and are powers of two.

However, it is possible to do it with lengths that aren't power of two in length. An old pi program of mine did that. It should also be possible



to work with numbers of different sizes, but that's not something that I've ever tried.

For  $a=4711$  and  $b=6397$ ,  $a_2=47$   $a_1=11$ ,  $b_2=63$   $b_1=97$  Radix=100

If we did this the normal way, we'd do

$$\begin{aligned} a_2b_2 &= 47 * 63 = 2961 \\ a_2b_1 &= 47 * 97 = 4559 \\ a_1b_2 &= 11 * 63 = 693 \\ a_1b_1 &= 11 * 97 = 1067 \end{aligned}$$

$$\begin{array}{r} 29 \ 61 \\ 45 \ 59 \\ 6 \ 93 \\ 10 \ 67 \\ \hline 30 \ 13 \ 62 \ 67 \end{array}$$

Or, we'd need  $N*N$  multiplications.

With the D&C method, we compute:

$$\begin{aligned} a_2b_2 &= 47 * 63 = 2961 \\ (a_2 - b_1)(b_1 - b_2) &= (47 - 11)(97 - 63) = 36 * 34 = 1224 \\ a_1b_1 &= 11 * 97 = 1067 \end{aligned}$$

$$\begin{array}{r} 29 \ 61 \\ 29 \ 61 \\ 12 \ 24 \\ 10 \ 67 \\ 10 \ 67 \\ \hline 30 \ 13 \ 62 \ 67 \end{array}$$

We need only 3 multiplications, plus a few additions. And of course, at longer lengths, additions are a lot simpler and faster than multiplications, so we end up ahead. Plus, we can do this algorithms recursively and when the numbers are very large, the savings is substantial.

Overall, the growth rate is  $O(N^{\log_2(3)})$  which is  $O(N^{1.585})$ .

It's worth pointing out that this method, unlike the 'transform' based



methods, doesn't have a failure point. It will work as long as you have the memory to hold it. The "transform" methods require your data type to be wide enough to hold the product and some extra bits.

Some example code is available on the [download page](#).

Most implementations require the length of the numbers to be a power of two. In the example program, I show how to allow the size to be any length. It should also be possible to allow the numbers to be different lengths, although I haven't coded that. (I've never needed it.)

[Home Page](#)[SiteMap](#)[Download page](#)[Feedback form](#)[▲ Multiplication](#)[Schoolboy](#)[Karatsuba](#)• [Transforms](#)[Big Digit](#)[Odd sizes](#)[Performance](#)

# Transforms

[FFT Limitations](#) | [FFT Types](#) | [Cyclic vs. NegaCyclic](#) | [FFT Styles](#) | [Background](#)

The 'transform' method is usually based on Fast Fourier Transform, although there are other types. The structures are pretty much the same, though.

Where as Karatsuba breaks the data into halves and then does three half sized multiplications, a FFT also divides the data into halves, but it multiplies one of them by a special "N'th" root of unity. Where as Karatsuba recursively breaks the data into smaller new chunks, a FFT just makes multiple passes over the same data.

I'm not going to go into the math because you don't really need it just for multiplication. At this point, you can really treat a FFT as a 'black box'. If you want to know some FFT background, go over to [Numerical Recipes](#). They have quite a bit of decent text, although the code they offer is (or at least was) extremely poor quality. Sections you'll probably be interested in chapters 12.0 through 12.4

Using a FFT based multiplication is a bit differently than either the Schoolboy or Karatsuba style. Those actually do the multiplication. A FFT is a 'transform' that pre- & post- processes the data and the 'multiplication' turns out to be just a simple vector convoltuion.

The steps for a FFT multiplication are:

- Load Number 1 into FFTData 1. Little endian first.
- Zero pad FFTData 1 so it is now twice as long.
- Load Number 2 into FFTData 2. Little endian first.
- Zero pad FFTData 2 so it is now twice as long.
- Do a forward FFT on FFTData 1
- Do a forward FFT on FFTData 2
- Do a convoltuion of FFTData1 and FFTData 2 into FFTResult
- Do an inverse FFT on FFTResult
- Scale the answer
- Convert FFTResult into normal integers. (If needed.)
- Release carries
- Store answer in 'Result'

All of the transform multiplication methods work this way. Some of the



specific details vary, of course, but the basic structure is always like this.

There are a few details that you need to notice, of course.

First, I said you had to put the numbers into the FFTData in little endian order. The least significant digit goes in FFTData element zero, etc. Technically, it is possible to do a transform in big endian format, but this is much less common and there is very little reason to even consider doing it.

Second, a convolution is just a simple element-by-element vector multiply.

Third, you need to scale the answer. Because of the way things work, each element will actually be multiplied by 'FFT Length' so we need to divide each element by that. Simple to do, of course.

Fourth, I said you may need to convert the answer to normal integers. Sounds odd, but if you do a FFT using regular floating point arithmetic (the most common way), each element will be a little off. It may be 37.9999 instead of 38. You need to round the answer to integer.

Fifth, I said you need to release the carries... That sounds a little odd. A FFT doesn't give the final product. Instead, it gives the "multiplication pyramid" To use the example I gave in Karatsuba, instead of getting the final answer like:

```

29 61
  45 59
   6 93
    10 67
-----
30 13 62 67

```

We actually get:

```

29 61
  45 59
   6 93
    10 67
-----
29 112 162 67
-----
30 13 62 67

```

Each 'digit' of our answer is actually the sum of the answers for that column. So the 'digit' can actually be larger than what it would normally be. So, we need to "release our carries" and get our normal answer.

And that's the basics of FFT based multiplication. There are a lot of additional details, of course. Like what style of FFT structure to use.

There are several ways to get to the same result. Some are more appropriate than others. And, of course, you have a choice of data types. You can do a plain ordinary floating point FFT, or you can do a modular math NTT (Number Theoretic Transform) or a Fast Galois Transform, etc.

But in spite of the numerous 'little details', the basic method is what I said above.

[As a side note, I pointed out that your data needs to go into the transform in "little endian" format. Least significant digit first. That's how it's done with a standard transform. It is possible to make a transform that is 'big endian', where you put the zero padding first followed by your data in big endian format. This can be a little more convenient, but it's uncommon. It's not hard to do. It's just a matter of changing the trig powers ordering (or changing the trig itself to be backwards) and changing the butterfly indexing (or changing the butterfly itself to be backwards.)]



[Home Page](#)[SiteMap](#)[Download page](#)[Feedback form](#)[▲ Transforms](#)

- **FFT Limitations**

[FFT Types](#)[Cyclic vs.](#)[NegaCyclic](#)[FFT Styles](#)[Background](#)

# FFT Limitations

FFT based multiplication is a wonderful thing because it allows us to multiply very fast.

However, like all silver linings, it has a big, angry dark cloud to go with it. There are actually several aspects. How you solve these will influence every aspect of your program. The way it looks, the way it behaves, the way it performs, how portable it is, etc.

The first has to do with the multiplication pyramid that I mentioned in the parent section.

Let's say you have X and Y and their product Z, and that each 'digit' is labeled as X(0,1,2,3,4,...N-1) Then the multiplication pyramid is:

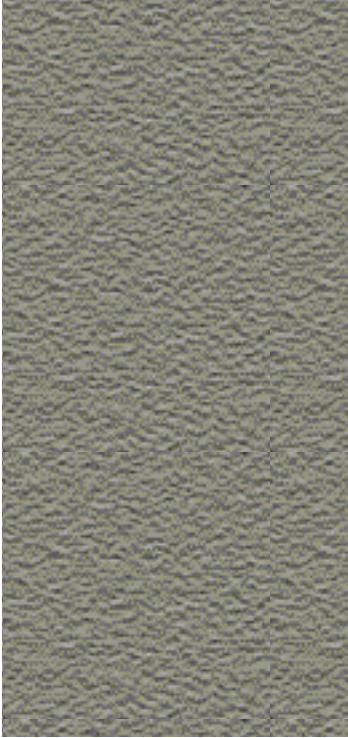
$$\begin{aligned}
 z(0) &= x(0)*y(0) \\
 z(1) &= x(0)*y(1)+x(1)*y(0) \\
 z(2) &= x(0)*y(2)+x(1)*y(1)+x(2)*y(0) \\
 z(3) &= x(0)*y(3)+x(1)*y(2)+x(2)*y(1)+x(3)*y(0) \\
 &\dots \\
 &\dots \\
 &\dots \\
 z(2n-3) &= x(n-1)*y(n-2)+x(n-2)*y(n-1) \\
 z(2n-2) &= x(n-1)*y(n-1) \\
 z(2n-1) &= 0
 \end{aligned}$$

It's sort of shaped like a pyramid, which is why it's called the 'multiplication pyramid'.

This is essentially the standard, normal, plain old digit by digit multiplication except you don't release your carries until after you get completely done.

This means that at its peak, the result Z can be far greater than our normal base. In fact, it can be as large as  $\text{FFTLengh}*(\text{base}-1)^2$ .

Because each element can be that large, it has to be able to hold it without overflowing. This can be dealt with, of course, it sometimes takes some creativity to do it fast enough to be worthwhile. A regular 'double' data type can quickly overflow. Other methods have their own strengths and weaknesses.



Another problem with FFTs are the amount of storage they consume. Depending on the type, it may be 8 bytes or more for each digit you multiply. So it's often necessary to be creative to reduce the storage consumption. A good consumption is 2 bytes for every decimal.

Even if you are creative, it may still be far larger than what will fit in memory, so you have to think about doing it using disk.

And the list really goes on. There are a whole series of problems when you want to do multiplication *really* fast. Sure, a FFT multiply is 'fast', but the choices you make can still result in a factor of 10 (or more) in runtime between a 'good' and a 'great' multiply.



[Home Page](#)[SiteMap](#)[Download page](#)[Feedback form](#)[▲ Transforms](#)[FFT Limitations](#)• [FFT Types](#)[Cyclic vs.](#)[NegaCyclic](#)[FFT Styles](#)[Background](#)

# FFT Types

[Floating point](#) | [NTT](#) | [Galois](#) | [Symbolic](#)

After reading the previous page, you have probably realized that a FFT multiplication is not a "one size fits all" multiplication method.

What type we chose will depend on what we need from it. If all we are needing is a few hundred thousand digits, then a simple [floating point FFT](#) is likely to be best. They are relatively easy to code and give decent results without excessive effort.

Beyond about half a million digits (maybe a million if you code carefully), either the data type is overflowing or it's taking too much memory (because we had to reduce the number of decimals per element in order to prevent overflow.) There are ways to increase the range, but it's often not worth the effort.

One alternative is a [Number Theoretic Transform](#) (or NTT) The structure is the same as a FFT except we work with modular numbers (ie: integers) instead of floating point. This is a bit more complicated and nearly guarantees the use of assembly language. It does, however, offer excellent memory consumption, good speed (if you are careful) and quite a bit of flexibility. For my own super large multiplications (billions), a NTT is the only reasonable choice.

A related method is a [Fast Galois Transform](#) (FGT). It's a cross between a floating point FFT and a modular math NTT. It's sort of like the two of them 'went behind the barn' together and 9 months later, a FGT was born. It does a 'complex' math FFT, except it uses modular integers. It has a little more flexibility and usability than a FFT, but it's not quite as good as a NTT.

The final type is a bit hard to describe. This is the [symbolic transform](#).

It's based on a FFT transform, except it does things in such a way that the transform itself doesn't need any multiplications. The multiplications are only needed for the convolution. I tend to call these transforms "fake FFTs" although they are usually called "symbolic FFTs". They are quite flexible. There are two types. The oldest is the [Schönhage-Strassen](#) multiplication, and the other is the [Nussbaumer convolution](#). They have the lowest known growth rate of any multiplication method. Unfortunately, real world implementations consistently show they don't perform quite as well as they should.

Implementations like in GMP v3.1.1 show that Schonhage-Strassen can be done fairly well, but it's not quite as good as their theoretical performance would suggest. No better than other methods.

Over all, if you are wanting just a few hundred thousand digits, use a [FFT](#). If you want more than that, use a [NTT](#) and be prepared for some effort and some assembly.

Of course, each of these have different [styles](#). There are a lot of ways to write a basic transform. And some are definitely more efficient than others. It's easily possible for one method to run at only half the speed of a better method.



[Home Page](#)[SiteMap](#)[Download page](#)[Feedback form](#)[▲ FFT Types](#)• [Floating point](#)[NTT](#)[Galois](#)[Symbolic](#)

# Floating point

[Complex](#) | [Real Value](#) | [Wrapper](#) | [Right Angle](#) | [Balanced](#) | [FFT Math](#)

A regular floating point FFT is, without a doubt, the most common transform around. It's widely used in all sorts of digital signal processing.

We can use several data types, but the most likely is a regular 'double'. How big a number you can multiply will depend on how many decimals (or bits) you put into each FFT element and how many extra bits are needed for good trig accuracy so we can recover the integer answer when we are done.

The best estimate that I've found was by Colin Percival. (His paper is available on the download page.)

With 'r' being the power of two for the length of the transform, the number of bits required is:

$$3.585 + \text{Log}_2(\text{Base}^2) + \text{Log}_2(r+1) + r$$

This is only an estimate, of course, but it's a reasonably decent one.

If we were going to put 4 decimals into each FFT data element and we were multiplying 1,048,576 digit numbers, the math would work out like:

1,048,576 / 4 dec = 262,144 number len  
262,144 \* 2 = 524,288 FFT length (zero padded.)  
'r' would be 19, since  $2^{19} = 524,288$

$$3.585 + \text{Log}_2(10,000^2) + \text{Log}_2(19+1) + 19$$
$$3.585 + 26.576 + 4.322 + 19 = 53.483 \text{ bits}$$

Since a regular 'double' data type is only 53 bits, you can see that we would be almost half a bit over. **However**, it might still work. Actually, it's likely that it will work.

In the formula, we were estimating we were actually putting our base into each element. In reality, we are only putting an average of 5,000 in there. That only takes 24.576 bits, which means it can all fit into a 'double' data type.

Plus, Colin's formula is a little conversative. (In fact, he actually gives



three levels. The 'worst case' based on the worst possible data, and 'average' error obtained from 'average' data.)

On the other hand, it also assumes that you have perfect trig accuracy. Which you aren't actually going to have. In fact, to reach 1m reliably, you may need to put quite a bit of care into your trig generation. It's a near guarantee that you'll have trig problems by the time you reach 1m decimals. The x87 FPU registers might be masking the problem, but it'll be there.

All in all, I suggest limiting yourself to just 512k decimals (4 decimals per 'double') and maybe 1m if you are sure of your programming and trig accuracy, or if you use some method to check the size of the multiplication error.

You might also want to use a "[balanced](#)" data format. That significantly reduces the average error rate. (Although the total potential error rate is nearly as high as normal.)

If you really want to push a FFT multiply, you can actually reach a point where you are multiplying 8 million decimals and getting a 16 million decimal answer, while still putting only 4 decimals per 'double'.

Of course, at that point, you are indeed pushing the FFT multiply far beyond it's theoretical limit. Pretty much a guarantee that it will fail during actual use.

A good way to deal with the possibility of error is to simply go ahead and do it and wait and see if the error occurs! If an error does occur, we use some other method. We can easily check for failure when we round our results to an integer before we release our carries. If it's too close to an integer, then we've lost too many bits and the results are questionable. If it's more than, oh, maybe 1/32 away from an integer, then we can't trust it.

You can, of course, put fewer digits per 'double'. However, the memory consumption starts becoming a problem. With 4 decimals, at 512k decimals, it takes 2 megabytes. At two decimals per element, multiplying 1m decimals would take 8 megabytes for each transform.

Rather than using a 'double', we could use some other data type.

A "long double" has 64 bits of mantissa. But it takes 10 bytes (usually 12 for data alignment). Not great. Sure, it'd work but you are only going to get just a little further with substantially more memory used.



We could use a "double-double" software package. Using two 'double' variables, this creates a 106 bit long data type. Keith Briggs has some in C and David Bailey has some in Fortran. The method works, and it lets us put 8 decimals into each element and reach about a billion decimals. All in all, though, it's a bit too much work for too little result.

(Plus there are some issues about the FPU. You've got be careful not to use the 'long double' mode of the x87, and make sure you are in the right rounding mode. And under Windows 9x, it's actually possible for the FPU precision mode and the rounding mode to change without your knowledge!! That happened to me a few times during some development work, and Colin Percival mentioned that it happened numerous times during his pi-hex project.)

You could also use a fixed point data format. Basically create your own data type that can hold hundreds of bits. Donald Knuth mentions this in his "The Art of Computer Programming" vol. 2, 3rd ed. He gives the 'bits required' formula of:

$$2^{(3K+2L+2.5+\lg(K)-M)} \leq 0.5$$

with  $K = \text{Log}_2(\text{fft\_length})$ ,  $L = \text{Log}_2(\text{number\_base})$ , and  $M = \text{data\_type\_bits}$ .

That means that for the 4 decimal digit example above,  $K=21$ ,  $L=26.58$ , and  $M$  would work out to be 97 bits. Quite a bit beyond the 53 bits of a floating point method.

So there is very little to recommend a fixed point method. Sure, it works. But it takes much more work to mess with our home made fixed point data type. I did try it once, out of curosity, and although I can't remember the exact level of performance, it was pretty unspectacular even allowing for a bunch of 'theoretical' optimizations I might have made.

The only reason I'm bothering to mention it at all is because Knuth talks about it. (This is a clear example of how something might seem useful to a mathematician while in reality us programmers know better.)

Doing a transform with wider data is expensive. The amount of effort to do the wider operations is considerable. Far, far beyond the cost of the 'atomic' operations that your CPU and FPU perform naturally.

The closest thing to a possible exception to this is it might be possible



for a carefully coded 'double-double' to run fast enough that your whole multiplication was only slightly slower. Some of those routines can pipeline fairly well, and allowing for the other overhead in a transform, it might work out about even.

As I say in the NTT section,.... stick with 'atomic' CPU and FPU operations.

Having said all of that, I really have to say that a FFT is only tolerable (and easy to write) up to about 128k or 256k digits. At 512k and higher you start encountering a lot of tiny details that can be difficult to program correctly. Just because it looks like it's reliable doesn't mean that it is.

Then throw in all the problems of doing it portably (including under Windows 9x, which has it's own problems), and it's often more trouble than it's worth. I myself have learned to hate doing any sort of floating point numerical calculation. Especially under Windows.

There has been some disagreement with my statements above. Namely, Dominique Delande showed that you can multiply up to about 128m decimals with 4 digits per element.

Well... yes. But there is a **significant** conditional. Namely that you are willing to accept the occasional failure of your multiplication routine! (Which also implies that you have to check each digit of the answer to make sure it is rounds correctly.)

For a 128m digit multiply, at 4 decimals per element, the transform would be 32m complex elements.

Even assuming 100% accurate trig, that's really pushing it. (And without 100% accurate trig, a terrifying risk. Every trig value will have to be accurate to within a couple of bits. Forget about using a standard trig recurrence. With the cosine value being 0.9999999xxxx those lower bits are critical. The sine value doesn't matter so much because it has useful data spread throughout the data. But the cosine will only have useful data in the lower part of the value, which is where the trig errors will accumulate.)



For a regular 0...9999 data, that's 26.575 bits just to hold the basic product, plus 25 for the pyramid (ie: sum of the products), for a total of 51.575. Only 1.5 bits for trig accumulation for conversion to integer. That is not very good.

Even switching to balanced -5000..+5000 will only get you two bits more. Just 3.5 bits for trig.

In my old text pi tutorial, I did say about 3 or 4 bits were needed for the trig, but that was based solely on experimentation. And I strongly recommended watching the error rate to make sure it worked, because at that range, you are pushing things a bit hard! You can't afford trig errors, round off errors, etc. This was based on me just putting 9999 into it and testing to see if it works. Well, since each element was the same value, that does allow a certain amount of sloppiness in the calculation.

Now, if you put balanced data into your FFT, and each digit is about 'average' (ie: somewhere about 5000, which in balanced form will be near zero), then you will be able to go much much further. Probably well into the billions.

But all of this has a cost..... Namely that you can't be sure of your answer. It's possible for it to be wrong!

Colin's paper gives a fairly solid mathematical analysis of the error rate and tells you how far you can go while still being able to **GUARANTEE** the correctness of the answer. He told me that it's slightly conservative, but it's still based on a solid analysis of the math.

His paper also gives a graph which shows some actual measurements of the error rate for 'typical' data and for some contrived 'worst case' data. The 'typical' error is significantly less, which means you can multiply much further. But the 'worst case' data has an error rate that is indeed much much closer to his theoretical error rate limits.

If your data happens to be 'typical' (or 'lucky') then you can multiply much further. If you have 'unlucky' data, then it will fail much sooner. The possibility of failure means you do have to check it's accuracy. (For example, when you round the floating point to integer, make sure that it's reasonably close to an integer already. Since it's supposed to be close to an integer, the further away it is, the more error has accumulated. If it gets beyond, oh, 1/16th or perhaps 1/32nd away



from being an integer, then you might want to be suspicious.)

So if you push things too far, you can't be sure in advance that it will be correct. You have to check and see. If it does fail, then you have to deal with it some how. Perhaps by using something such as Karatsuba / Fractal multiply to break it into smaller chunks and then redoing the whole multiplication.

And even if it is wrong, it might not matter! In many situations, a little bit of error isn't significant. In some cases, it just doesn't matter. In other cases it may be corrected later. For example, the Newton routines (ie: square root and division) are self correcting. If a small error occurs, it will correct itself later. I've encountered several cases myself where my multiplication routine was sometimes wrong but I was still getting the right final answer.

Now.... if you want to take that risk, well, that's your choice. Just be sure and tell people that your results are suspect. (Unless you've verified them with a second independant calculation using a different algorithm, or verified them against a known good answer.)

But there isn't really a heck of a lot of reason to do so. A NTT can run about as fast and can guarantee a 100% accurate answer. And you don't have to mess with all the floating point math issues under Windows. And you don't have to deal with the issues of some "doubles" being 'double' size and sometimes being 'long double' size while in the FPU registers.

(Or if you want to, you could just use a [Schönhage-Strassen](#). The math program GMP shows that it can be done fairly competitively. Although it only works in binary and not decimal.)

I stand by my statements above. If you want to be **SURE** of your calculation, then you need to use a method that is provably correct and that isn't going to fail occasionally.

In my early pi programs, I didn't know or care. I got the right answer and that was all I cared about. I played "fast and loose." After learning a bit more about floating point math and how it might fail (and seeing it fail on occasion), now I do care. I believe going with a provably correct method is the only reasonable choice. There are already numerous chances for failure during a calculation.

(Programmer error, compiler error, hardware error, cosmic radiation



flipping random bits, etc.) Why deliberately add yet another? Cutting corners is okay sometimes, such as round-off errors accumulated during a Newton routine. Other times, errors aren't acceptable.

Dominique was willing to take the risk and I'm not willing to take the risk. That's really what it all comes down too.... the amount of risk you are willing to take, and the amount of time you're willing to occasionally spend redoing the multiplication when the FFT fails.

(For the record, Dominique did verify his answer with a known good answer, so his final result was correct.)

[Home Page](#)[SiteMap](#)[Download page](#)[Feedback form](#)[Floating point](#)

- **Complex**

[Real Value](#)[Wrapper](#)[Right Angle](#)[Balanced](#)[FFT Math](#)

# Complex

Most FFTs are of a 'complex' nature. Meaning they have both a "real" and an "imaginary" part.

The easiest way to use one in a multiplication is to put our data into the 'real' part and zero out the 'imaginary' part. (And of course the FFT data array should be zero padded.)

We then do the transforms, convolution, and inverse transform. And we get our data out of the 'real' part and ignore the imaginary part.

The convolution is a straight element by element multiply. Just a simple:

```
for (x=0; x < Len; x++)
  FFTNum1[x]=FFTNum1[x]*FFTNum[2];
```

Nothing to it. (This is 'complex' math, and the single multiply shown above is actually a full 'complex' multiply.)

We scale the answer, round it, and then we get the data out in the same 'little endian' format that we put it in.

Of course, this is a little inefficient. Half the data array (the 'imaginary' part) is simply ignored.

There are four solutions.

The first is to take advantage of the math and put two data sets into the FFT. One in the 'real' and the other in the 'imaginary'. I'm not going to bother discussing this further because it's not a very good solution. Sure, it works. If you are doing two transforms, it's twice as fast. But it can still be faster. And the inverse transform (after the convolution) will only need one, so we still pay the full bloated price of that method. If you are still interested, go check the [Numerical Recipes](#) site.

The second choice is to use a [Real Value](#) transform. Those are a little hard to come by, although the [Fast Hartley Transform](#) (FHT) is 'real' only. I've never been very fond of the FHT. It does work, and it works tolerably well. But I just don't like the structure. It has to be done just



slightly differently. It's actually quite a bit more complicated. I personally never got great performance from it, although others told me that in their tests it ran slightly faster.

The third choice is to use a '[wrapper](#)' around a regular transform. That tricks the 'complex' transform into thinking its working with complex data. In reality, it's pretty much an optimized FFT pass. You are doing basically the same math as if you were doing a plain FFT. You just aren't needing the extra memory.

The fourth choice is to use a relatively new method called a [Right Angle](#) transform. Instead of doing any fancy math to trick the FFT into thinking it's working with 'complex' data, we instead put the data in a special way, and scale the data. It's pretty simple. It has only one slight 'problem' and in my opinion is probably the best choice, with the possible exception of a FHT (although I personally don't like FHT.)

Wrappers are more common, and it's what I used, but now that I know about a right angle, I consider it to be better.

Demonstration code is available on the [download page](#).

[Home Page](#)

[SiteMap](#)

[Download page](#)

[Feedback form](#)

▲ [Floating point](#)

[Complex](#)

• **Real Value**

[Wrapper](#)

[Right Angle](#)

[Balanced](#)

[FFT Math](#)

# ***Real Value***

[FHT](#)

I don't really have a lot to say about a 'Real Value' transform.

They aren't genuine real value FFTs. They've just been modified to automatically work with the data knowing it has a bunch of zeros.

I've never seen a nice, readable, descriptive real value transform. All of the ones I've seen have been very complex. That's the result of the optimizations to make it a 'real value' transform.

That means I can't give you an example. Sorry.

You can, however, go to the [Fast Hartley Transform](#) page and get a real value transform. The FHT works just as well as a "real value" transform.

The 'Hartley' transform is very similar to the 'Fourier' transform. A little more difficult to program, but it might be worth it for you. And it's inherently a "real value" transform.

The error rate for a FHT seems to be less than for most Fourier transforms because there's no need for a conversion between 'real' and 'complex'.



[Home Page](#)[SiteMap](#)[Download page](#)[Feedback form](#) [Real Value](#)• **FHT**

# ***FHT***

The Hartley Transform was discovered by R.V.L Hartley back in 1942.

The 'fast' version of it seems to have first shown up in a book by Ronald Bracewell in 1984. In spite of the Hartley transform being published 42 years prior, and the techniques to convert a Fourier Transform into a Fast Fourier Transform (which also apply to the Hartley Transform) also dating back 40 (or more) years, Bracewell decided to patent the algorithm. If anybody wanted to use the algorithm, they had to pay Bracewell a royalty. Suffice to say, that wasn't popular. (Ahhh, the joys of the U.S. patent system.)

Fortunately, with a bit of persuasion, Bracewell came to his senses a few years later and didn't renew the patent. (Or, the cynics among us figure he just forgot about it until right after it expired, at which point it was too late to renew.) It's now in the public domain. (Although many people still mistakenly assume it's patented, resulting in the FHT being avoided.)

It's a real value transform. The basic structure is indeed very similar to a Fourier transform.

It is a little more difficult to implement, though.

I don't want to get into a lot of math, so instead, I'll give you a couple of references. NumRec, unfortunately, doesn't have anything on the FHT, but the background on the FFT is pretty much the same as the FHT. The techniques to convert a discrete Fourier Transform into a Fast Fourier Transform also apply to the discrete Hartley Transform.

Back in April 1988, BYTE magazine had an article by O'Neil on the FHT. The only other major article that I know of was Robert Scott's article in Embedded Systems Programming, in September 2000.

The FHT is done very much like the FFT. They are somewhat related and you can translate the data between the two using a simple wrapper function.

Where as the FFT works with the 'butterfly' which uses roots based on natural log (as done by cosine & sine in a complex field), the FHT works with a 'CAS' function in the real field.



The basic structure of a FHT is pretty much the same as a FFT. You can do recursive, iterative, DiT, DiF, radix-2, radix-4, split radix, etc. etc.

Unlike the FFT, the FHT doesn't really have a 'direction'. The FHT uses the same cosine and sine values regardless whether it's doing a forward or an inverse transform. The FFT requires a +1 or a -1 in the trig generation, of course.

The FHT is a little harder to program, though.

The FFT takes two data elements, does the butterfly on them, and then puts them back in their own spots.

The FHT doesn't do that. It takes two separated data points and generates a single point. If you do it 'in place' like you would a FFT you would overwrite your data before you got finished with it.

You've got to either use a separate work area (which takes too much space) or you have to do two points at once.

Here are a couple of DiT style recursive FHT's. Compare them to simple recursive ones on the [DiT](#) FFT page. The scrambling has already been removed and would need to be done before you called either of these functions.

In the first one, notice the required work space just to be able to do the 'CAS' function?

```
void RFHT(double *a,int n)
/* Recursive Fast Hartely Transform.  Requires work
space */
{int x;
 double theta;
 double OutPut[MAXSIZE*2];

n/=2;
if (n>=2) RFHT(a, n);
if (n>=2) RFHT(a+n,n);
theta=M_PI/n;
for (x=0;x<n;x++)
    {double cas1,cas2,t;
     int i=n-x;
     if (x==0) i=0;
     cas1=a[n+x]*cos(x*theta)+a[n+i]*sin(x*theta);
```



```

        OutPut[x]  = a[x] + cas1;
        OutPut[n+x]= a[x] - cas1;
    }
    for (x=0;x<n*2;x++) a[x]=OutPut[x];
}

```

In the next one, see how we have to do two separate points at once so we can get rid of the working space?

```

void RFHT(double *a,int n)
/* Recursive Fast Hartely Transform. */
{int x;
 double theta;
 double temp;

n/=2;
if (n>=4) RFHT(a,  n);
if (n>=4) RFHT(a+n,n);
theta=M_PI/n;
if (n==2)
    { /* Do the two simple 2 point transforms.*/
        temp=a[0];
        a[0]=temp+a[1];
        a[1]=temp-a[1];
        temp=a[2];
        a[2]=temp+a[3];
        a[3]=temp-a[3];
    }

/* Do the special k=0 loop below */
temp=a[0];
a[0]=temp+a[n];
a[n]=temp-a[n];
for (x=1;x<n/2;x++)
    {double cas1,cas2,t;
      int i=n-x;
      cas1=a[n+x]*cos(x*theta)+a[n+i]*sin(x*theta);
      cas2=a[n+i]*cos(i*theta)+a[n+x]*sin(i*theta);
      temp  = a[x];
      a[x]  = temp + cas1;
      a[n+x]= temp - cas1;
      temp  = a[i];
      a[i]  = temp + cas2;
      a[n+i]= temp - cas2;
    }
}

```

```

    }
    /* Now do the n/2 point */
    if (n/2)
    {double cas1;
      int x=n/2;
      int i=n-x;
      cas1=a[n+x]*cos(x*theta)+a[n+i]*sin(x*theta);
      temp  = a[x];
      a[x]   = temp + cas1;
      a[n+i]= temp - cas1;
    }
  }
}

```

For use in multiplication, you've got a couple of choices. You can either use a wrapper around the FHT to make it look like a FFT, or you can work with the plain FHT results.

My demo code shows both methods because you need to know about the differences. Although a FHT works much like a FFT, it's not identical.

For a FFT wrapper style, the data points end up in a different order than what you would expect. The real is in the low half and the imaginary is in the high half. So, the convolution has to be done differently.

For the plain FHT style, if you do it as a DiF/DiT pair (to avoid the scrambling) the convolution is a lot more difficult because the FHT convolution needs data points that are separated. You've got to do the first two. Then the next four. Then the next eight. Then the next 16. Then the next 32. Etc. (Yeah, it took me a little while to figure out. When I investigated the FHT, all I had was the old BYTE article and one confusing FHT implementation which gave me something to compare my results to.)

The error rate for a pure FHT (no wrapper) recursive transform is pretty good, though. Significantly lower than a 'wrapper' style regular FFT.

Example code is available on the [download page](#).



[Home Page](#)[SiteMap](#)[Download page](#)[Feedback form](#) [Floating point](#)[Complex](#)[Real Value](#)• [Wrapper](#)[Right Angle](#)[Balanced](#)[FFT Math](#)

# Wrapper

Most 'real value' transforms are actually a regular 'complex' transform with a Real / Complex wrapper around it.

Most wrappers are solely for the DiT style transform. I think I once had a DiF style somewhere, but I can't find it anywhere. I may have accidentally deleted it during some cleaning. Or maybe I'm wrong and I didn't have one.

The wrapper is almost a regular FFT pass over our data. It pre-processes the data, taking our 'real' data, using some imaginary zeros for the complex part, and putting it into a regular 'complex' form for our regular FFT. A bit more to it than just that, of course, but that's pretty much the principle.

The data format is a little differently, though.

Complex elements one through the end are done normally, but element zero is different.

If you had done a full 'complex' FFT of the full data (putting the data only in the real and zeroing the complex), element zero and element  $\text{Len}/2$  would both be 'real', with zero in the imaginary. And elements  $\text{Len}/2+1$  through the end would be a conjugate of elements 1 through  $\text{Len}/2-1$ .

Since the upper half is just the conjugate of the lower half, we can ignore it. Since elements zero and  $\text{Len}/2$  are both real, we can put both of them together.

So, the final 'real' data format ends up being

```
Result[0].Real = FFT[0].Real
Result[0].Imag = FFT[Len/2].Real
Result[1] through Result[Len/2-1] = FFT[1] through FFT[Len/2-1]
FFT[Len/2+1] through FFT[Len/2] ignored.
```

This means the convolution has to change slightly.

```
FFTNum1[0] = Cmplx(real(FFTNum1[0])*real(FFTNum2[0])
                    imag(FFTNum1[0])*imag(FFTNum2[0]));
```

```
for (x = 1; x < Len; x++)
    FFTNum1[x] *= FFTNum2[x];
```

The core of the wrapper is

```
for (i = 1, j = Len - i; i < Len/2; i++, j--)
{
    Cmplx p1,p2,t;

    CmplxConj2(t,Data[j]);
    CmplxAdd(p1,Data[i],t);
    CmplxSub(p2,Data[i],t);CmplxMul(p2,p2,PRoot);
    /* Tricky. Swap and change sign. */
    CmplxSet(t,-Dir*p2.i,Dir*p2.r);

    CmplxAdd(Data[i],p1,t);
    CmplxSub(Data[j],p1,t);CmplxConj(Data[j]);

    /* Normalize */
    CmplxDivV(Data[i],2.0);CmplxDivV(Data[j],2.0);

    NEXT_TRIG_POW;
}
```

although you wont find it quite like this very often. Most implementations will explicitly expand those operations and then fold them together to get better performance.

Demonstation code is available on the [download page](#).



[Home Page](#)[SiteMap](#)[Download page](#)[Feedback form](#) [Floating point](#)[Complex  
Real Value  
Wrapper](#)[■ Right Angle  
Balanced  
FFT Math](#)

# ***Right Angle***

A "Right Angle" transform is a way of weighting the data so you can put your real values into a complex FFT without having to do a wrapper. (A 'Right Angle' is actually a special case of a Discrete Weighted Transform.)

Instead of doing a wrapper and changing the FFT, you change the data.

- You put your data into the real part of the complex variable and zero the imaginary. You don't need to explicitly zero pad because the 'imaginary' part of the variables will be used.
- You then scale the data using a very simple trig recurrence.
- Then you do the regular complex FFT
- You do a simple complex vector mul
- You do the inverse complex FFT
- You do the inverse scaling
- You then release your carries. The Real part has the first half and imaginary has the second half.

This method allows you to use a DiF followed by a DiT style transform without doing the scrambling.

It's the simplest form of Real<->Complex wrapper there is. The old style in the previous section is more common, but this is much easier. And more efficient. It is supposed to have a lower error rate, which means there's less chance your multiplication will be wrong due to rounding errors.

My example program doesn't have a very good error rate. It has quite a bit of error in the trig recurrence part. (Of course, it is only sample code.)

Demonstration code is available on the [download page](#).





[Home Page](#)[SiteMap](#)[Download page](#)[Feedback form](#) [Floating point](#)[Complex](#)[Real Value](#)[Wrapper](#)[Right Angle](#)• [Balanced](#)[FFT Math](#)

# Balanced

A 'balanced' transform is not an actual way of doing a FFT multiplication. Instead, it's a way of putting the data into the FFT. It can be used with any of the other methods.

Normally, we put the data into the FFT so it's positive. From zero to our base. However, with a balanced transform, we allow it to be negative. Half the range is below zero and the other half is above zero.

That spreads the error out a little and makes the multiplication a little more reliable.

There isn't a lot of advantage to it, actually. It does reduce the error rate a little, and because the data is signed it takes up less space, but for multiplication, there isn't a lot of advantage.

With 4 decimals per element, a FFT mul is reliable up to 512k decimals. With care, you can do 1m decimals reliably enough. Two million and higher is too big.

A balanced data format would improve things and make 1m and 2m digits reliable (assuming accurate trig), but beyond that it would still be unreliable and would still likely take too much memory.

So, really, it's only useful for the 1m and 2m range. Less than that and you don't need it. More than that and the FFT is taking too much space and the data would still overflow anyway.

I do have some sample code in my multiplication demos on my [download page](#).

[Home Page](#)[SiteMap](#)[Download page](#)[Feedback form](#)[Floating point](#)

[Complex](#)  
[Real Value](#)  
[Wrapper](#)  
[Right Angle](#)  
[Balanced](#)

- **FFT Math**

# FFT Math

There are only a few bits of math that you need to know.

First, the standard complex data format is based on the old FORTRAN complex data type. A real followed by the imaginary. Whether you do it as a FORTRAN or C++ data type, or a C struct, or just simply as a pair of data elements in an array, that is the standard format.

Some people like to do the real and imaginary separately. However, that just adds yet another thing to keep track of, and it also doubles the number of data accesses that have to occur because when they are together, both are brought into the CPU's cache with a single reference. (Assuming you have the data aligned properly.) And having them apart like that increases the 'cache thrashing' because the variables try to occupy the same cache location (due to the way most caches are designed.)

A basic 'complex' multiply is:

```

R_r=(X_r * Y_r) - (X_i * Y_i);
R_i=(X_i * Y_r) + (X_r * Y_i);

```

It is possible to do it with just 3 multiplications. In fact, there are a couple ways.

```

Temp=X_r*(Y_r + Y_i)
R=Temp - Y_i * (X_r+X_i)
I=Temp + Y_r * (X_i-X_r)

R=(X_r * Y_r) - (X_i * Y_i)
I=(X_r + X_i) * (Y_r + Y_i) - R

```

However, it's not all that useful. Back in the 60's when floating point multiplication was extremely expensive it made sense. But with modern processors, you aren't likely to encounter a need.

(For the following discussion, I'll call the root 'Root' and the powers 'Pow'. Those variables are usually called 'w' and 'u' for some reason. I do it myself sometimes. But 'Root' and 'Pow' are easier for you to understand.)



We compute the root of unity using:

```
Root=exp(Cmplx(0.0,(M_PI*Dir*(2.0))/step));
```

However, you rarely actually see it that way. The reason, of course, is that the EXP() function is a little inaccurate, and more importantly, it results in too much error later in the trig recurrence. (The speed of the exp() function is irrelevant. We are only doing it  $\log_2(\text{FFTLen})$  times. Unless we are doing a recursive transform, of course.)

Instead, we generally compute it using explicit sine & cosine calls.

```
Root_r=sin(M_PI/((LENGTH)*2));
Root_r=-2.0*Nth_r*Nth_r;
Root_i=sin(M_PI/(LENGTH))*(DIR);
```

The reason isn't because of our dislike of the exp() function but because it helps us generate more accurate powers (Pow) of our root. More on that later.

Sometimes you may see it done a little differently.

Such as using pre-generated tables.

Or one of several methods where you replace the sin() calls with a relationship to compute the new value based on the old value.

Or perhaps based on some Nth root of unity. A N'th root of unity is basically the same except it's precomputed for the largest transform we might do, and then we compute the current root based on that. This assumes the Nth root is difficult to compute or the author just likes the looks.

The simplest way to generate our root powers would be to just do something like:

```
Pow=Pow*Root;
```

Pretty simple. (Or if you wanted to be descriptive, like some of the code in the [DiF](#) & [DiT](#) sections, you might even use the pow() function.)

However, except in example programs, you won't ever see it done this



way. **Never**. The reason is fairly simple... It accumulates errors too quickly. You can only do short transforms before the errors get so bad your results are wrong.

During the transform, both the real and imaginary parts of the root (variable 'w') will range from nearly zero to nearly one. When it's close to zero there is no problem because all those leading zeros aren't stored with the floating point data type. Just the actual useful data is stored. But when it approaches 1.0, it starts becoming 0.999....9xxxxx... All those leading 9's take up space in the data type, even though they add little useful information. Only the x's (in the number above) contain actual useful data. But those get pushed off the end.

Instead, we can be clever. We take advantage of the cosine & sine relationship and manage to store only the useful digits and then when we compute the 'Pow' variable, we combine it in a special way that gives us the same answer but with less accumulated error.

In other words, since  $\text{Root}_i = \sin()$  like normal, but  $\text{Root}_r = 0 - \sin()$  instead of  $\text{Root}_r = \cos()$  and because of that, all those 9's aren't stored and we have a more accurate trig value. And that means we have to compute the trig recurrence a little differently.

We use the trig recurrence:

```
temp = Pow_r;
Pow_r = Pow_r * Root_r - Pow_i * Root_i + Pow_r;
Pow_i = Pow_i * Root_r + temp * Root_i + Pow_i;
```

This is very much like a standard complex multiply. Just the final addition on the end is different. But that little change (and the change in our Root variable) makes all the difference.

It's not 100% accurate, though. There is still a little bit (about half a bit, actually) of error accumulation in the recurrence and it does build up. Sometimes other forms are better. For example, you may want to regenerate the values (via sine & cosine) every 32nd time or something. There are even other methods to ensure trig accuracy.

You shouldn't have to do that unless you are really pushing the limits of a FFT. Frankly, by that time you'll be running into quite a few other FFT based multiplication limitations. I did that once, for various reasons, and I have to admit, it wasn't really worth the effort.



I will say, however, that it's extremely easy to write non-portable code when using the PC. The reason is the 'long double' FPU registers. Without even trying, you can get the benefits of the longer registers. And that significantly improve your trig recurrence accuracy. If you need portability, whether to a different processor or even to a different compiler, you have to be a lot more careful.

Even compiling with different options (such as optimizations, or debugging, etc.) can cause changes in what the compiler keeps in the FPU registers. Problems like that can be hard to track down.

If you are concerned at all about portability, try using a compiler such as the old Watcom which completely disables 'long double'. With other compilers, such as Microsoft C or GNU C, you can change the FPU mode but you can't guarantee that the compiler won't change it later or even that the libraries will work properly. It probably will be okay, but unless you check carefully, you can't be sure. It's possible for some library routine, an interrupt or the OS to change it without you even knowing.

Changing the FPU precision or mode under Windows 9x is a little risky, actually. Sometimes Windows changes it back! Various .DLL's (including the dll for the standard C runtime library that ships with Windows) can change it when they load. This is indeed documented by Microsoft. I spent a lot of time tracking the problem down in my program, only to discover that several other people already knew about it and that Microsoft & Borland warned about it.

(Things like the automatically changing FPU settings are the reasons I got so tired of using FFTs or even using the FPU to do the NTT modular math. Integer is more reliable. And another reason to stick with Linux or old DOS for your calculations.)

The more I worked with floating point math (under Win9x) the more I hated it. There are so many places where you can lose accuracy, where you can do something wrong, and where Windows itself conspires against you. I reached a point where I flat out hated the idea of doing any FPU operation. Sure, the FPU is fast, but if you can't be sure of the results, then what good is it?

As I learned more, I began to realize how lose I had previously played with accuracy, and how lucky I had been to get the right answer

All these reasons are why I prefer to work solely with NTTs, using integer operations that you can depend on.



[Home Page](#)[SiteMap](#)[Download page](#)[Feedback form](#) [FFT Types](#)[Floating point](#)• [NTT](#)[Galois](#)[Symbolic](#)

# NTT

[Mod math](#) | [Special Primes](#) | [Wide NTT](#) | [Multi-prime](#)

A Number Theoretical Transform (NTT) is sometimes known as a Pollard transform, a prime modulus (or radix) transform (PMT or PRT), or an integer transform. I think a NTT is more accurate. I particular dislike calling it an "integer transform" because we work with modular numbers, not integers, and you can do a regular FFT using integers (as fixed point numbers, with manual scaling & shifting.)

It's really very much like a regular transform. The differences are that instead of using a floating point, you use an integer. More speicifically, a modular integer. Instead of having a N'th root of unity using trigonometry (sine & cosine) you use a special prime number and operate modulo that.

Why would we actually want to use a NTT though? Well, to get around the limitations of a floating point FFT. Remember all those 'bits used' in the previous section? And the amount of memory used? A NTT can much more easily go much further without taking nearly as much memory.

Simple as that. Well.... not exactly simple. There are some problems that I'll discuss shortly.

Since there is no inaccurate trig for us to deal with, our 'bits required' formula is much simpler. It's just:

$$\text{Log2}(\text{Base}^2) + \text{Log2}(\text{FFTLen})$$

Pretty simple, huh?

(It's worth pointing out that with a FFT, the data can overflow at any point, and the results be wrong. With a NTT, the only time it can overflow is after we've done the transform, after we normalize it (by dividing by NTTLen), and are about to release the carries. That's the only time. All the rest of the time the data is deliberately overflowing. That's what modular arithmetic is all about.)

So... Let's see just how far we can go. Hmmm.. Just by looking at it, I can see that a 32 bit integer is just not going to be big enough. A 64 bit number is going to be the minimal.

If I put 4 decimals per element, like before, I'd be able to reach:



$64 - 26.57 = 37.43$  Or just 37, since we don't need the fractional bits.  
And since our prime number isn't going to be a full 64 bits, let's just say 36 just to be safe.

So, that means we could do a transform of  $2^{36}$  That's 64 billion long.  
With each element holding 4 decimals and allowing for zero padding, that means we'd could multiply up to 128 billion decimals!!!

**WOW!** Simple example code is available on the [download page](#).

The memory usage would be Decimals/4\*2\*8 which works out to needing 4 times as many bytes as decimals. Not bad. Not bad at all, actually. (Although I mentioned before that I could get it down to half that, or just twice the number of digits in storage. Sometimes that extra savings is indeed important.)

But there is more to think about than how much memory it takes.  
Decent run time would also be nice!

So, the question is... How can we do a modular multiply quickly? For that, we head to the [ModMath](#) section.

Another issue that's worth pondering is how can we reduce the memory? As I said above, sometimes you need the reduced memory consumption. When you try to multiply numbers that are billions of digits, you may flat out not have the disk space to spare.

The big waste in a NTT is the pyramid size. The  $\text{Log}_2(\text{FFTLen})$  part.  
No matter how many decimals we put into each element, it will always cost that much. So why not put more digits into each element?

Let's see... If I still wanted to multiply up to 128 billion decimals:

$$4 \text{ dec} = \text{Log}_2(\text{Base}^2) + \text{Log}_2(\text{FFTLen}) = 26.576 + 36 = 62.576$$

$$\text{Cost} = 15.644$$

$$8 \text{ dec} = 53.151 + 35 = 85.151 \text{ Cost} = 10.644$$

$$16 \text{ dec} = 106.302 + 34 = 140.302 \text{ Cost} = 8.769$$

$$32 \text{ dec} = 212.604 + 33 = 245.604 \text{ Cost} = 7.675$$

Well...it looks like it gets pretty big rather quickly. Notice, though, that the 'bits per decimal' cost has dropped. It starts out at 15.6 and drops to half that.

Hmm... Frankly, it sounds rather difficult to multiply numbers that are



246 bits wide. Well, it is. There are several ways to deal with that.

The first option is to just go ahead and do it with a regular multiply.

Hardwire it for that size to reduce overhead, etc. And chose a special prime to make the modulo easy. This isn't really a very good choice because the multiplication itself is going to take a little longer than you'd like and because there aren't any good primes that would make the modulo fast.

Another way would be to make the number wide enough so you could use a little fft multiply on it! Using a 'discrete weighted transform' can make the modulo 'automatic'. This is similar to how George Woltman does things in his GIMPS (Great Internet Mersenne Prime Search) program. But that doesn't work well either. By the time the number is wide enough to do this, it's going to cost you too much for the multiply and for the overhead.

The wider the number, the more work it takes to multiply. That grows faster than any savings we would get from doing a shorter transform.

So we've got to have as narrow a number as possible! But yet to save storage we have to make it as wide as possible. <sigh>

We could use a montgomery modular multiply (discussed in [ModMath](#)) which would make the modulo easier, but that wouldn't help the multiplication cost itself.

There is a solution, and that's to use a [multi-prime NTT](#) with a Chinese Remainder Theorem. This lets us break a wide NTT into several narrower transforms and then build up the answer as if we had done the wide NTT. Although it does have a cost... Namely a near guarantee that you'll have to use assembly language.

Doing a multi-prime NTT is a lot more difficult than a regular NTT.

And doing a regular NTT is more difficult than a FFT. It's harder to do the math efficiently.

[Home Page](#)[SiteMap](#)[Download page](#)[Feedback form](#)[NTT](#)

- [Mod math](#)
- [Special Primes](#)
- [Wide NTT](#)
- [Multi-prime](#)

# Mod math

[ModMul](#) | [33-64 bits](#) | [Montgomery](#) | [Other ModMul](#)

Doing modular math is pretty easy. Doing it fast is a little more difficult, of course.

Generally it's easier to work with modular numbers that are positive. You can do them 'balanced' where they are both positive and negative, but things are easier (and work just as well) when they are positive.

Addition and subtraction are pretty easy, of course.

```
ModInt ModAdd(ModInt Num1, ModInt Num2, RawInt
Prime)
{RawInt Sum;
Sum=Num1;
Sum=Sum+Num2;
if (Sum >= Prime) Sum=Sum-Prime;
return Sum;
}
```

```
ModInt ModSub(ModInt Num1, ModInt Num2, RawInt
Prime)
{RawInt Dif;
Dif=Num1;
Dif=Dif+Num2;
if (Dif < 0) Dif=Dif+Prime;
return Dif;
}
```

(RawInt is large enough to hold the sum/dif of the operation. If it isn't, you have to be more careful and check for over/under flow.)

Doing a ModMul() is a little more difficult. Basically we can do something like this:

```
ModInt ModMul(ModInt Num1, ModInt Num2,RawInt Prime)
{LONGLONG Prod;
Prod=Num1;
Prod=Prod*Num2;
return (ModInt)(Prod % Prime);
}
```



And as long as "LONGLONG" is big enough to hold the full product and the modulo operation works, then this will work.

For other methods of modular multiplication, see the section on [ModMuls](#). To keep this page short, I'll just blindly assume you can do a ModMul somehow.

To do a NTT, we need three semi-constants and a [special prime](#). For the sake of brevity here, I'll assume you already have the special prime and its associated primitive root.

Each of these semi-constants can be computed on the fly. The computational cost isn't great. Some people prefer to precompute them, use tables, etc.

The first semi-constant is our root of unity. This is derived from the prime, the primitive root, and the length of our transform. (If you want to precompute the Nth root and then build it up during the transform.)

```
NthRoot=ModPow(PrimvRoot,Prime-1-(Prime-1)/NTTLen);
```

Or, if you compute it 'on the fly' during the transform (comparable to the cosine & sine calls in a floating point FFT), you would do it like:

```
Root=ModPow(PrimvRoot,Prime-1-(Prime-1)/Step);
```

Doing it "on the fly" as needed is a bit cleaner, I think. The ModPow only takes a few modular multiplies and it's only called  $\log_2(\text{NTTLen})$  anyway.

The second semi-constant is our inverse root of unity. It's what we use to do an inverse NTT. (Remember the 'Dir' variable in the complex floating point FFT's? We can't do it like with a NTT, so we need to compute separate semi-constants.)

```
NthRoot1=ModPow(PrimvRoot,(Prime-1)/NTTLen);
```

The 'on the fly' version is:

```
Root1=ModPow(PrimvRoot,(Prime-1)/Step);
```

The third semi-constant is our multiplicative inverse for our transform length. Remember in the complex floating point FFT we had to divide



our answer by FFTLen? With modular numbers you can't easily divide. You can however multiply by the inverse. This isn't genuine division, just cancellation, but that's all we need.

```
MulInv=FindInverse(NTTLen);
```

FindInverse is defined as:

```
FindInverse(x)=ModPow(x,Prime-2);
```

And of course, modular powers are done with a standard 'binary' power method.

```
ModInt ModPow(ModInt base,int expon)
{ModInt prod,b;
  int x;

  if (expon <= 0) return 1;

  b=base;
  while (!(expon & 1))
  {
    b=ModMul(b,b);
    expon>>=1;
  }
  prod=b;

  while (expon>>=1)
  {
    b=ModMul(b,b);
    if (expon & 1) prod=ModMul(prod,b);
  }
  return prod;
}
```

These semi-constants can be precomputed, stored in tables, etc. (which is why I call them "semi-constants"), but they are easy enough that you can compute them whenever you need them.

I should comment on the 'normalization' done in the ModAdd() and ModSub(). You may see it done a little differently, in order to remove the conditional statement. Modern processors (and especially the poorly designed Pentium-4) lose a lot of performance when they mispredict a jump. And these operations will be nearly random. 50% of the time the processor will mispredict the jump.



Anyway, sometimes you'll see it done as with the Shift & Mask method to correct for underflow.

$$\text{Num} = \text{Num} + ((\text{Num} \gg 31) \& \text{PRIME})$$

If the number is negative, it shifts the sign bit over the entire word (making it all bits set) and then it masks the prime, making the expression equal to Prime. If the number is positive, then no bits get shifted and the mask ends up being zero.

This method is restricted to a max of 31 bit primes with 32 bit words, or a max of 63 bit primes with a 64 bit word. In other words, signed data.

[Home Page](#)[SiteMap](#)[Download page](#)[Feedback form](#) [Mod math](#)

- **ModMul**
- [33-64 bits](#)
- [Montgomery](#)
- [Other ModMul](#)

# ModMul

As was pointed out in the parent section, doing the modular multiplication is the hard part.

Sure, we can do something like this:

```
ModInt ModMul(ModInt Num1, ModInt Num2, RawInt Prime)
{
    LONGLONG Prod;
    Prod=Num1;
    Prod=Prod*Num2;
    return (ModInt)(Prod % Prime);
}
```

And as long as "LONGLONG" is big enough to hold the full product and the modulo operation works, then this will work. On the X86, it'll work for 32 bits or less, since the x86 generates the full 64 bit answer of a multiply. Other processors (such as the PowerPC) have to do more work.

There are several problems with that method

First, the modulo operation is slow. It takes close to 40 cycles on most processors. Trust me, that's not very fast. Especially when you realize that the multiply operation took only a couple of cycles.

Second, if we are working with primes that are larger than 32 bits, then obviously we can't code it like that. If you are working with 64 bits you might want to check [here](#). You might also want to check the section on [Montgomery modular multiplication](#). That kind of modular multiplication makes doing the modulo very easy.

It's also possible to do a generic method.

If you are working with primes that are 32 bits or less, then your problem is speed.

One way to do this is to do this is to use both the FPU and the integer part. The FPU computes the high part and the integer part computes the low part. We then multiply by the reciprocal to do the division.

```
double GLOBAL_INV=1.0/Divisor;
```



```

long GLOBAL_DIV=(double)Divisor;

long timesmod( long a, long b)
{
    /*This is a clever 31-bit modular multiply that
    doesn't
        need 64-bit math. Note that the number to
    reduce by
        (and its double precision reciprocal) must be
    stored
        in globally. For details see ACM SIGPLAN
    notices,
        vol. 27 no. 1, p.95 */

    double dquot = (double)a * (double)b * GLOBAL_INV;
    unsigned long reducedquot = dquot + .5;
    unsigned long remainder = a * b - reducedquot *
    GLOBAL_DIV;
    return (remainder & 0x80000000? remainder+n:
    remainder);
}

```

This is very portable. It'll work on any processor. Another way to write it is:

```

ModInt
ModMul(ModInt a, ModInt b)
{ModInt rem;
rem = a * b;
rem = rem - Prime * ((ModInt) floor(0.5+RecipPrime *
((double) a) * ((double) b)));
if (rem < 0) rem +=Prime;
return rem;
}

```

Both of these are limited to 31 bits, because they need to allow for the chance of the intermediate result being negative.

The big problem with this method is that it's slow, especially on the x86. First, the conversion from 'double' to integer is extremely slow on the x87.

Second, it doesn't pipeline well. On modern processors, it's critical to keep the processor as busy as possible.

Third, if you try to code this in assembly, you have to make sure you use the same rounding mode as what C uses. This is a somewhat minor point, but I mention it because I once spent quite a while debugging some code because elsewhere in the program I had changed the rounding mode. Under C it didn't matter because it always set the rounding mode and then changed it back when it got done.

Another way is to take advantage of the x87's "long double" registers, which can hold a full 64 bits. The powerPC and most other processors don't support this, so this is definitely not portable!

You can do this with code such as:

```
#define MAGIC 3*2^63
#define JUSTIFY 3*2^52

double temp;

void ntt586mul( long *a, long *b, long size ) {

    register double f0, f1;
    long ALU, temptr = (long *)&temp;

    do {
        f0 = (double)a[0];
        f1 = (double)b[0];
        f0 = f0 * f1;
        f1 = f0;
        f0 = f0 * reciprocal;           /* f0 = quotient, f1 =
remainder */

        f0 = f0 + MAGIC;
        f0 = f0 - MAGIC;               /* round to nearest by
justifying mantissa */

        f0 = f0 * prime;
        f1 = f1 - f0;                 /* compute a*b-n*q */

        f1 = f1 + JUSTIFY;             /* push answer to bottom of 53-
bit mantissa */
        temp = f1;                   /* store to temporary space */

        ALU = *temptr;                /* put bottom 32 bits of
mantissa into ALU */

        ALU = ALU + ((ALU>>31) & prime); /* if answer
```



```

negative add prime */
    a[0] = ALU;                                /* put remainder
back */

    a++; b++; size--;
} while(size);
}

```

There are several clever things going on here. First, we are using the x87's registers to compute the full 64 bit product. (Actually 62 bits, because the x87 can only load signed 31 bit integers.)

Second, we are being clever in how we round our result. (The FPU **MUST** be in 'long double' mode and in 'round' mode.) Since the x87 doesn't have a nice fast 'round' instruction, we do it ourselves by adding 0.5 to it. (Actually, this is taken care of by the FPU being in 'round' mode.) When we add a very large number to it that causes the FPU to 'round' the number because it's so big that it's pushing the fractional bits right out of the register. It's overflowing in a controlled manner. We then subtract that same value and we have our answer rounded.

Third, we are avoiding the slow double->integer conversion that plagues the x87. I don't know why Intel never made it faster, but they didn't. It is hideously slow. So slow in fact that it's better if we go out of our way and do it ourselves. We do this by adding a 'magic' number that pushes our answer down to the lower 32 bits. We then store the 'double' and then access the lower integer part directly. And we have our answer.

Fourth, we are being clever in how we normalize our answer. It could actually be below zero, in which case we need to add PRIME to it. We just check the sign bit, right shift it all the way down (which creates a mask of 0x00000000 or 0xffffffff), 'bit and' it with the prime. If it was negative, the value equals prime. If it was positive it's equal to zero. We then add that to the answer.

This is just a fancy way of saying: if (prod < 0) prod+=PRIME; The advantage is there isn't a jump / conditional in it which can cause the pipelines to flush and a computational 'stall' to occur. On many processors (especially old ones) it's faster to just do the explicit 'if' statement check.

Now, you can do this in C. I've done it. However, it's a little touchy with aggressive compilers. Sometimes the constants get removed



because it notices you are adding and then subtracting the same value. The solution is to put each access into a separate variable declared as 'volatile'.

Also, many compilers are very sensitive to you loading the integer part of the double you just stored. It may notice it's not needed and not bother to actually store it. And it may be sensitive to the style of code you write to retrieve the integer part. GNU C is sensitive about this, even going so far as to ignore certain ANSI/ISO C requirements.

All in all, this method works a lot better when you write it in assembly language!!

Another reason to write it in assembly language is that with very careful coding you can get several results in the same time it takes to get just one! [Jason Papadopoulos](#) once wrote some pentium optimized code for me to do this, and he manage to get it down to just 64 cycles for 4. About the same time that it would take to get just one answer. His public domain code is available on the [download page](#).

But what if you don't want to do it like this? What if you aren't running on a x86 or are working with primes larger than 31 bits?

In that case, you've got a couple of choices.

If you are using 64 bit primes, check [here](#).

For general purpose stuff (including 62 bit primes), check the section on [Montgomery's modular multiplication](#).



[Home Page](#)[SiteMap](#)[Download page](#)[Feedback form](#)[Mod math](#)[ModMul](#)■ [33-64 bits](#)[Montgomery](#)[Other ModMul](#)

## 33-64 bits

Doing a 64 bit modular multiply is not easy when all you have is a 32 bit computer. Even if you have a 64 bit computer it might still not be easy to obtain the 128 product and do the modulo.

There are two solutions. Use [Montgomery modular multiplication](#) for multidigit numbers, or use special primes.

Some primes are easier to do the modulo than others. For example, the 64 bit primes:

```
FFFFFFFF00000001
FFFFFFFFC0000001
FFFFFFFF000000001
```

are slightly easier to do as a modulo than most. This is because there are so many bits set in the upper half.

However, I would still not call it easy.

Mikko Tommila's APFloat package has a wide range of modules, one of which uses the three 64 bit primes above. The file [Raw.h](#) from the module 64unix.zip gives an example of how it can be done. It's not pretty. It is better than not being able to do it at all, and better than many other methods, but it's not pretty. (This file is distributed under Mikko Tommila's freeware license which allows modification and distribution provided it's not sold.)

This method does require only four 32 bit multiplications, but it requires a lot of other operations too. And it's pretty much limited to just the three primes above. Maybe a few others, but they become a bit harder than those three.

If you can live with 62 bit primes, you can use the method described in the [Big Montgomery](#) section. It shows a fairly clean way to do it in 6 integer multiplies, but there is less overhead. It does require 64 bit integers. (All GNU C compilers supply them. As doe all C99 standard C compilers.) It does have the advantage that you can use many more primes than just the three above.

Another possibility is to use primes that are about 50-52 bits. Those will fit into a 'double'. You then use a method similar to the 'timesmod()' method in the previous ModMul page. Except, you use a software based 'double-double' package to create a 106 bit LDouble type to do the math with. (Both [Keith Briggs](#) and [David Bailey](#) have such packages.)

It definitely works. But it is a bit slow and awkward. To get good performance, you'd probably have to try and pipeline it and do two at a time. It would also use more memory than an integer method. Plus you have to deal with the same kind of FPU problems as in with a floating point FFT. The FPU is just too unportable and too unreliable for my taste.

I've never actually done a full NTT this way, although I think Xavier Gourdon used this method in his pi program.



[Home Page](#)[SiteMap](#)[Download page](#)[Feedback form](#) [Mod math](#)[ModMul](#)[33-64 bits](#)■ **Montgomery**[Other ModMul](#)

# Montgomery

[Big Montgomery](#)

Montgomery multiplication is a way of doing a modular multiply without doing a slow modulo operation. Instead, you convert the modulo into a multiplication and the division ends up being just shifting. It converts a difficult modulo into an easy modulo.

To use Montgomery, you preprocess all the data (convert into Montgomery space), do your NTT, and then post process all the data (convert out of Montgomery space.)

Addition and subtraction work the same as before with the same prime.. The one area that is different is the modular multiply.

By changing the order of the modular numbers you are working with, Peter Montgomery realized that you could change the division / modulo into division / modulo by a power of 2, which is just a shift.

It isn't free, though. You do have to do some math to do the modulo, but it's easier math.

There isn't a lot on the web about Montgomery multiplication. Well, there is but most of what is available is geared towards cryptography, and often hardware implementations. They have conditions we don't, so their solutions aren't entirely suitable for us.

However, there are a few items of use. The first is an old posted message that gives a reasonable decent explanation of a simple implementation. [Montmul.txt](#)

There are some other references, but they are oriented toward [big Montgomery](#) multiplication, where the numbers don't fit into a single word.

And with the working example code I'm including, you should be able to understand the basics of doing Montgomery Modular Multiplication.

(Note that there are many ways to do this. Several 'flavors' plus research is continuing to develop new methods. What I'm showing here works and that's good enough for you right now.)

The first thing we need to do is decide on what we want our 'easy divide' to be. Our radix 'R'. This is usually the next power of two larger than



your prime. For example, on a 32 bit computer, you should probably use  $2^{32}$ . Your word size. That makes division very easy. (Remember, I said Montgomery modular multiplication makes a hard modulo into an easy modulo. The modulo is still there.)

We then need to find three constants. 'ToMontC', 'FromMontC' and 'MontMulC'. (I've appended a 'C' to the function names where they are used. Makes it easier to remember.)

(An existing ModMul has to be used. It's only used a few times, so you can be sloppy and do it with a 'bit by bit' method.)

ToMontC is obviously the constant we use to convert to Montgomery representation. To convert to Montgomery we ModMul this constant and our data. Of course, since our ModMul is slow, it's better to convert the constant to Montgomery and that way we can do a MontMul with our new constant and our data.

The actual formula to compute ToMontC is

**ToMontC = -(Prime<sup>-1</sup>) mod R**

Assuming that you chose 'R' to be your word size, that works out to just

**ToMontC = (0 - Prime) % Prime;**

FromMontC is used to convert from Montgomery representation. Again, you ModMul it and the data to get the new (regular) data. And again, since that means using a slow ModMul, it's faster to convert 'FromMontC' to Montgomery form and then MontMul it and the data to get the original data.

Computing FromMontC is easy. We simply find the regular modular multiplicative inverse of our original ToMontC value. Which is just:

**FromMontC = ModPow(ToMontC, Prime-2)**

Our last constant we need is the MontMulC, which is used during the MontMul itself. This constant is the multiplicative inverse of our prime, except it is in Montgomery format based on our radix 'R'.

It's defined as:

$$RR' - NN' = 1$$



Or,  $R'$  is the inverse of  $R$  (our base) under multiplication mod  $N$  (or prime), and  $N - N'$  is the inverse of  $N$  under multiplication mod  $R$ .

Again, this is an 'inverse' so and we could use

**MontMulC=MontPowR(Prime,Prime-2);**

(MontPowR is a regular binary power routine, like ModPow, except we work modulo our word base, instead of a prime. If our base is a regular 32 bit integer, we would just be doing regular 32 bit multiplies, getting the regular 32 bit answer.)

But actually it works out that you can just say:

**MontMulC=Prime-2;**

So why bother 'computing' anything?

Now, what is the actual Montgomery multiplication function?

The algorithm is normally given as a 'reduction' function. In other words, just the Montgomery Modulo part. It assumes you've already multiplied the two 32 bit numbers and gotten the 64 bit answer.

Here it is:

```
function REDC(x)
  m=((x mod R) * MontMulC ) mod R
  t=(x + m * Prime) / R
  if (T < N) return t;
  else return Prime-t;
```

'R' is just our word size, so it's just selecting the high or low words. (Unless you wanted to be 'funny' and chose a different base.)

Assuming we are using 32 bit primes, and haven't done the multiply yet, an actual implementation is:

```
ModInt MontMul(UINT32 Num1,UINT32 Num2)
{UINT32 Lo1,Hi1,Lo2,Hi2;
  UINT64 LP1,LP2,T;
  UINT32 m;
  LP1=( (UINT64)Num1 )*( (UINT64)Num2 );Lo1=LP1;Hi1=LP1>>32;

  m=(Lo1 * MontMulC);
```

```
LP2=((UINT64)m)*((UINT64)Prime);Lo2=LP2;Hi2=LP2>>32;

T=((UINT64)Hi1)+((UINT64)Hi2);
if ((Lo1+Lo2) < Lo1) T++;

if (T >= Prime) T-=Prime;
m=T;
return m;
}
```

That may not be the clearest. The code does use 64 bit integers even though the primes are only 32 bits. That's just the way Montgomery Multiplication works.

This is actually pretty much a waste of time, though.

Let's be blunt here... If you were using 31 or 32 bit primes, there are faster ways to do things. Jason's Pentium optimized 31 bit ModMul springs to mind.

**BUT** where Montgomery shows its advantage is when you do primes that are larger than your word size.

For that, you need to see the sub-section, [BigMontgomery](#).



[Home Page](#)[SiteMap](#)[Download page](#)[Feedback form](#) [Montgomery](#)

- **Big Montgomery**

# Big Montgomery

What if we want to do 'big' montgomery. For example, what if we are using 64 bit primes.

We could just do it normally, using full 64 bit math operations, but it's more efficient to treat the montgomery multiply as if it was a multi-precision number, composed of two 32 bit integers. Which it is on a 32 bit computer.

The best reference I found was a paper by Cetin Koc, Tolga Acar, and Burton Kaliski Jr. titled "Analyzing and Comparing Montgomery Multiplication Algorithms. It's available on the [download](#) page.

As you may know, there are a couple ways to do basic schoolboy multiplication. You can scan across the numbers and generate the result a piece at a time, or you can generate each digit of the product completely. (Product scanning is unusual. The slow verify in my multiplication demos use it, if you are curious.)

By threading the Montgomery reduction into those methods, the authors come up with 5 different algorithms to do a big number Montgomery multiplication. The paper is more concerned with big numbers for cryptography, but it works just fine for our two word 64 bit number, although their performance results aren't applicable.

If you read their paper, you'll probably wonder which method to use. They pretty much recommend all the methods but for different purposes.

I found their "FIPS" (Finely Integrated Product Scanning) method to be the most useful due to a combination of operations count, overhead, and storage consumption. You may disagree, of course.

For a two word number, the operations work out to:

```
x=0;
x+=Num1[0]*Num2[0];
Prod[0]=(x * MontConst) % WordSize;
x+=Prod[0]*Prime[0];
x/=WordSize;
x+=Prod[0]*Prime[1];
```



```

x+=Num1[0]*Num2[1];
x+=Num1[1]*Num2[0];
Prod[1]=(x * MontConst) % WordSize;
x+=Prod[1]*Prime[0];
x/=WordSize;
/* loop 2 */
x+=Prod[1]*Prime[1];
x+=Num1[1]*Num2[1];
Prod[0]=x % WordSize;
x/=WordSize;
Prod[1]=x % WordSize;
x/=WordSize;

```

With 'x' being a 64 bit number. (This method limits us to 62 bit primes, so we can have two bits left over for the carries. Not really a problem.)

As expected, it does require 10 multiplications. Integer multiplications, which are usually slow on most processors.

But the formula can be improved.

First, we need to realize that MontMul allows the top word to have a different number of bits used than the rest. This means the low half can be a full 32 bits and the upper word consume only 30 bits. We don't have to put 31 bits in each. That makes things a lot simpler, since extracting the high or low word becomes trivial.

The second thing we can do is take a look at the Prime[0] What if we could get rid of that? It'd get rid of two multiplications. Well... Since Prime[0] is the low word of our prime, we can't exactly get rid of it.

Unless by some chance, the low word of our prime is '1'. After all, multiplication by '1' doesn't need to be done.

And that would leave the 30 upper bits of our prime to actually be some number. Yeah, we can do that. All we have to do is simply chose our prime with a low word of '1'. That gets rid of two multiplications and leaves us with just 8.

Anything else... How about the upper part of the prime... Hmmmm.... Not really. It's gotta have some information in it. If we chose our prime right we might be able to get by with a shift and sub, but that probably isn't going to be very quick. No quicker than just doing the multiply.



So what is that "MontConst" in the formula? Of course, that's the regular Montgomery multiplication constant we need. But what value is it? Is it something we can work with.

If you compute the constant for a few 62 bit primes, you notice something amazing! It's "all bits set"! Due to our decision that all the primes have a low word of just '1', the constant turns out to be a genuine constant!

That means we can hardwire it. After all, a multiplication by 'all bits set' really just means we are negating the other number.

We end up with a 62 bit Montgomery Modular Multiply routine something like this:

```
ModInt MontMul(ModInt Num1,ModInt Num2)
{UINT64 Sum,P00,P01,P10,P11,T64;
  UINT32 A0,A1,B0,B1,Prime0,Prime1;UINT32 T;

  A0=MONTLOW(Num1); A1=MONTHIGH(Num1);
  B0=MONTLOW(Num2); B1=MONTHIGH(Num2);
  Prime0=MONTLOW(Prime);Prime1=MONTHIGH(Prime);

  P00=MUL64(A0,B0);
  P10=MUL64(A1,B0);
  P01=MUL64(A0,B1);
  P11=MUL64(A1,B1);

  T= MONTLOW(-(UINT32)MONTLOW(P00));
  T64=MUL64(T,Prime1);

  Sum=P00+T;Sum=MONTHIGH(Sum);
  Sum=Sum+T64+P01+P10;

  T= MONTLOW(-(UINT32)MONTLOW(Sum));
  Sum=Sum+T;Sum=MONTHIGH(Sum);
  Sum=Sum+MUL64(T,Prime1);
  Sum=Sum+P11;

  if (Sum >= Prime) Sum-=Prime;
  return Sum;
}
```



Just 6 multiplications. No matter what method we chose to do a modular multiplication, we would absolutely positively need 4 of them just to do the basic multiplication itself. Doing the modulo is only costing us two extra. And there is no messy comparisons etc., like in the method Mikko Tommila uses with his 64 bit primes.

How well does it run... Well, that'll depend on your C compiler, of course. (And your processor, of course. A 486 or even a classic Pentium isn't going to run it well. But there are not many people left with either of those.)

I had some tests done with various C versions and an asm version on various systems, and on a Pentium-II an asm vector version came out to about 40 cycles per MontMul. A C coded version came out with about 62, and GNU C is infamous for poor 64 bit "long long" code.

That really isn't too bad at all. Jason P.'s super fast vector modular multiplies for 31 bit primes takes about 16 cycles per prime. That's about 32-34 cycles for 62 bits worth of work. (That's significant because Jason's 31 bit modular multiply is the fastest code on the net. You'd be hard pressed to find any other 31 bit modular multiply as fast, much less faster.)

The MontMul assembly is much much \*MUCH\* easier. Pretty much a straight translation of the C code, just without the register shuffling that GNU C likes to do.

Even in C, though, the performance is pretty good. Throw in the overhead for the other operations, etc., and you are going to have a lot of trouble finding anything significantly better.

The final conditional normalization can be done straight inline, just like I mentioned on the Mod math page.

```
Sum=Sum + ((Sum >> 63) & PRIME)
```

At this level of performance, memory latencies may very well cost enough to kill your performance. Overhead for individual operations will cost too much and you'll have to do everything as vectors just to reduce the overhead.

And a good thing about this code is that we no longer have to mess with the FPU. It doesn't matter what precision it is (64 or 80), what rounding mode, or even whether Windows decides to change it on a whim, without telling you about it (which *can* happen!!) (I can't even



begin to tell you how much I dislike floating point under Windows 9x! [shudder])

I gotta admit, I'm pretty happy with this kind of code. It's not perfect. It might not be the fastest. But you'll be hard pressed to find anything faster. It compares very well with Jason's high performance 31 bit code. Which was extremely tuned for the Pentium FPU.

It's what I had decided to use in my unfinished v2.7 pi program. (It's unfinished for other reasons. I was happy with the MontMul.)

By now, you may be wondering how far we can go with this.

For 4 decimals into it and use a single prime, the 'base square' size will be 26.58 bits. That leaves 35 bits for the pyramid. Allowing 1 bit for the zero padding, that means you can do  $4 \cdot 2^{34} = 64g$  decimals. And it takes  $4 \cdot \text{Decimals}$  bytes of storage.

For 8 decimals in two primes, the base will be 53.16 bits, leaving 70 bits for the pyramid. Suffice to say, that's more than you could ever possibly do....

For 16 decimal, in two primes, the base size is 106.31 bits. Leaving nothing for the pyramid size.

For 32 decimals in four primes, the base size is 212.61 bits, leaving 35 for the pyramid. Allowing for the zero padding, that means  $32 \cdot 2^{34} = 512g$  decimals. And it takes  $2 \cdot \text{Decimals}$  bytes of storage.

You could go to 8 primes, but what'd be the point?

You could modify the MontMul() I show above to handle 63 bit primes, but you wouldn't be gaining much. Still, if you needed to, you could. I'll leave it as an exercise for the reader...[grin]

With a bit of care, you can even use a 64 bit prime if you limit upper word of the prime to less than `0xFFFFFFFF0`, it'll still all fit into a regular 64 bit 'long long' integer, although much more care is needed in the routine, and you have to do the multiplies as high/low halves, etc.

The 62 bit version is simplest, though. And it doesn't require the FPU.

A simple example of the 62 bit MontMul in x86 assembly is available on the [download page](#).





[Home Page](#)[SiteMap](#)[Download page](#)[Feedback form](#) [Mod math](#)[ModMul](#)[33-64 bits](#)[Montgomery](#)■ [Other ModMul](#)

## Other ModMul

Other than the stuff I've mentioned for word size primes, and for 33-64 bit primes, and for general Montgomery ModMul, there aren't many useful options left.

Your only options left are

- 1) Just tolerate it.
- 2) Find some special prime that you can work with efficiently. Sort of how the 64 bit primes were done.
- 3) Use a Mersenne Prime with a FGT. At least then the modulo is easy. The FGT isn't the best algorithm, and the selection of Mersenne primes is a bit limited.
- 4) Use a special Discrete Weighted Transform multiply to automatically do the modulo. A lot of overhead. Only suitable for really big primes.
- 5) Use some smaller prime!

Number four sounds interesting.... This is a variation on Crandall's regular DWT. Colin Percival came up with it and described it to me. I never did implement it, though.

I seriously thought about it, but while running a bunch of 'theoretical timings tests', which I counted the operation count for various sized transforms using various ideas and potential optimizations (a lot quicker than writing the code and spending weeks tweaking it!) I realized that it still suffered the same problem as a regular wide NTT.

Remember, the effort to do a wider data element grows faster than the savings of having to do a shorter NTT. Even though the DWT uses a FFT, it still has a growth of  $O(N \cdot \log_2(N))$ . For small 'N' (like the DWT would use), the growth is pretty substantial. On the other hand, for long 'N' (like our total NTT length) grows / reduces rather slowly.

The effect is the wider you make the data, the slower the program will run.

Plus, there is extra overhead in doing a DWT modular multiply.

Realistically, though, there is little reason to use anything beyond a 62 bit prime because with the Chinese Remainder Theorem we can always build up the NTT result as wide as we need.



[Home Page](#)[SiteMap](#)[Download page](#)[Feedback form](#) [NTT](#)[Mod math](#)

- [Special Primes](#)

[Wide NTT](#)[Multi-prime](#)

# Special Primes

You may wonder what kind of primes are "special".

Well... The key requirement for a NTT is that it has a Nth root of unity. So that means it has to be at least as large as the length of the transform you are wanting. Makes sense.

There are a few other conditions that I wont discuss, but it ends up that we have to use a prime of the form:

$$1+K*2^x$$

So our prime has to be some multiple of some power of two, plus one... Not too difficult. Obviously  $2^x$  will be at least as big as the length of the biggest transform we are going to do, so we just loop though K's and check for primality.

Of course, since  $2^x$  in the prime must be at least as large as the size of the transform we are doing, that puts a few restrictions on the size of the primes, how many primes, etc.

Once we've found our prime we need a "Primitive Root of Unity". A "primitive root" is just a fancy way of saying that when you raise it to a power, it just goes through all of the numbers. Without degenerating into some constant.

That's a little more difficult to compute than the prime itself.

Factor (Prime-1) into its parts. We then compute

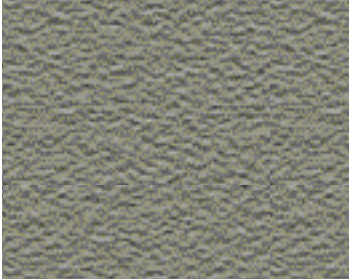
$$R^{((Prime-1)/Factor)} \neq 1 \text{ Modulo Prime}$$

for all the factors of (Prime-1) computed above.

If we find any value of R that passes those tests, then it's a primitive root. You can just try  $R=2, 3, 4, 5, \dots$  and you'll find one fairly quickly.

If you want to be absolutely sure,  $\text{ModPow}(\text{PRoot}, \text{Prime}-1)$  should be 'one'. And if you really, really want to be sure, just loop through the entire range, and Prime-1 should be the only one equal to 'one'.

Here's an old program that I wrote to find these primes. It's pretty



simple. It's not the fastest, but it's a one time computation so it doesn't matter. It computes the primes, the root and the inverse.

[Download page.](#)



[Home Page](#)[SiteMap](#)[Download page](#)[Feedback form](#) [NTT](#)[Mod math](#)[Special Primes](#)• [Wide NTT](#)[Multi-prime](#)

# Wide NTT

By 'wide' NTT I mean do it where the prime number is big enough to hold everything you need.

This may be 128 bits or 256 bits or a thousand bits, etc. Just some size that is greater than what you can do with the built in math in your CPU.

Well... my advice is **DON'T**.

The problem with doing wide transforms is that it's harder to do the math. The add, the subtract, and especially the multiply and modulo.

Sure, it sounds good. Just do the data wider. But how are you going to do that wider modular multiply?

The wider the multiply, the more effort it takes. The same thing applies to small scale stuff like this as it does to really big multiplies like these entire pages are talking about.

How are you going to efficiently do the multiply? Sure, it's only a 'few' bits, but it still takes a lot of effort.

That amount of effort grows quicker than any savings you might get from doing a shorter transform.

For example, let's assume we use a small FFT to do the modular multiply. (For small sizes, a regular schoolboy would be faster, but I'm talking general growth here, and a FFT would have a smaller growth.)

Let's also assume you can instantly load the data and extract the results (which is certainly not going to be true). The growth would be the standard  $O(N \cdot \log_2(N))$  of a FFT.

If you've ever computed the growth for a variety of 'N', you'll notice that small sizes grow much faster than the larger ones do.

The effect is that because of the modular multiplication, the cost of doing a wider NTT grows faster than the savings gained by having to do a shorter NTT.

Even with various optimizations, such as using Colin Percival's DWT



method, and using a hardwired FFT with no loop overhead, etc. etc., the growth of the small FFT will be that above.

If you actually implement this, you will find a "sweet spot" where extra cost of the wide modmul balances the savings of the shorter NTT.

The exact point will depend on your coding skills.

Based on the theoretical operations counts I did, it would probably be somewhere between 32 and 128 decimals. Say 64 just for the sake of discussion. Below 32 and the cost of doing the FFT is too great.

Above 128 or so and the growth has likely outweighed the savings of the shorter NTT and your efficient coding skills.

For the smaller sizes, you could use some other multiply and do the modulo the hard way, but that too takes time.

For that size, frankly, we can do a multi-prime NTT with a comparable amount of effort, if not less effort.

When I first learned about doing a wide NTT with the DWT, I was real excited about the possibility. However, the theoretical costs and actually trying to implement it was disappointing. Loading the data into the FFT, doing the DWT scaling, doing the hardwired optimized small transform, doing the DWT inverse scaling, releasing the carries... All of that takes time.

And I'm not that good of a coder. Even being optimistic, there was no way I could get it to perform as well as other methods. Your coding skills may be better.

Everything that I've experienced says that things run most efficiently when the data types are closest to what the processor can handle.

That means 32 bit integers, floating point types, and sometimes 64 bit integers (even on a 32 bit computer).

About anything larger than that is going to have too much overhead.

Except for a few special situations, my advice is to not do a 'wide' transform. Do a [multi-prime](#) transform instead.

But, if you want to see an example of a very simple version, I've got one in my multiplication demos on my [download page](#).





[Home Page](#)[SiteMap](#)[Download page](#)[Feedback form](#) [NTT](#)[Mod math](#)[Special Primes](#)[Wide NTT](#)■ **Multi-prime**

# Multi-prime

[CRT](#)

I guess the first question you have is why should you do a multi-prime transform. Doesn't it take more effort and more time?

Well, yes it does take more effort. Quite a bit more. It might even take a little assembly to do it efficiently.

So why do we want to do it? Two reasons. First, by putting more digits into a NTT element, we can cut the memory consumption in half.

That can be pretty important. Second, it's easier to do several transforms with narrower data than it is to do one with wide data. (In other words, it's easier to work with 32 bit data than it is 64 bit data.)

But it doesn't have to take more time. In fact, it may be slightly faster.

If you just do a wider transform (either by multiple primes or with a wider prime), then it's going to take more time. If it takes 1 minute to do something, then doing two of them will take 2 minutes. If you double the width, it's going to take at least twice as long. It's just common sense.

But, you don't have to do the exact same size transform. Remember in the parent page, I gave the example of increasing the number of digits per element? Well, that's the key to doing a multi-prime NTT.

You increase the number of digits and the number of primes, rather than just one.

If we put twice as many digits into each NTT element, then that means the transform is only half as long. So each one is twice as fast. But since we have to do twice as many transforms, it evens out.

So, how can we use multiple primes to do a NTT???

We take advantage of the [Chinese Remainder Theorem](#).

As you may know, the CRT allows you to combine several modular numbers (like we are already using) and get the same answer as if we had done it with a much larger modular number.

The basics are really not hard.



In the parent NTT section, I gave the example of putting four decimals into each NTT and using a 64 bit prime.

In this example, I'm going to say 16 decimals and four 32 bit primes. (Why 32 bit primes? Simple... because they are easier to work with on a 32 bit computer! I could use two 64 bit primes.)

Here are the following steps:

- For each prime, modulo each group of 16 decimals and put it into the NTT.
- For each prime, perform a forward NTT.
- Repeat steps 1 & 2 with the second number to multiply.
- For each prime, do a convolution between the two transformed numbers.
- For each prime, do an inverse NTT.
- For each element in the four transform, do a CRT over those four data items. You'll get a 128 bit answer (in this example) out of the CRT.
- Release your carries from the 128 bit wide data.

The problems with doing a multi-prime NTT are doing the last two steps. The wide CRT and releasing your carries from the wide data.

That often requires some assembly to do it efficiently. (At least on the register poor x86.)

Some demonstration code is available on the [download page](#).

One of the reasons to use a multi-prime NTT is so you can use smaller primes. You don't need to work with enormous numbers.

Well, there is a disadvantage to using small primes. Such as 31 & 32 primes. Namely, their small size!

Yeah, that's right. Their small, easy to work with size is both an advantage and a disadvantage.

The disadvantage is in the size of the transform they can do. After all, their root depends upon the power of two they are based on. And with smaller primes, there is a very definite limit to how big that can be.

If you use the 8 largest 31 & 32 bit primes, you can put 32 decimals into each element and multiply a billion decimals. That's as far as you can go. The length of the transform depends on what the prime is built on, and those 8 primes have reached their limit. You can't add more primes (such as doing a 16 prime NTT) because there aren't any more.

You've reached the limit of the 32 bit integers.

If you want to go beyond that, you'll have to use larger primes. No way around it. How much larger will depend on which approach you chose. Like I discuss in the [Mod Math](#) section, there are several sizes to choose from and several ways to do the modular multiplication.



[Home Page](#)[SiteMap](#)[Download page](#)[Feedback form](#)[Multi-prime](#)

• CRT

# CRT

Doing a Chinese Remainder Theorem is not all that hard. The hard parts are:

- Dealing with the short multi-precision math.
- Deciding which one to use
- Discovering a few items that are 'obvious' but are not always clearly mentioned.

Let's take the first point... Well, if you are needing this, then you probably already know the basic algorithms and can easily code a few simple big integer routines. If you are lucky enough to have a compiler that generates good code, you may be done. If you are like the rest of us, you may need to write a few core routines in assembly.

The second point is deciding which CRT to use. Knuth mentions two, and another is described in Mikko Tomilla's APFloat package and in Joerg's HFloat.

Personally, I prefer to use the plain one Knuth describes. It's fairly easy to implement and runs tolerably well.

The formula he gives is:

```

v[1]=u[1] mod M[1]
v[2]=(u[2]-v[1])c[1][2] mod M[2]
v[3]=((u[3]-v[1])c[1][3] - v[2])c[2][3] mod M[3]
....
v[r]=(((u[r]-v[1])c[1][r]-v[2])c[2][r]- ... - v[r-1])c[r-1][r] mod M[r]
```

And then to combine them:

```

u=v[r]*M[r-1]*...*M[2]*M[1]+ ... +v[3]*M[2]*M[1] + v[2]*M[1]
+ v[1]
```

And the constants are done with:

$$C[I][J] = 1 \text{ (modulo } m[J])$$



The first part of the algorithm is just simple modulo math. Simple stuff. It can be done with just a couple of nested loops using the same modular math that your NTT etc. uses.

The second part, where we combine the answer into the whole, is the only part where we have to work with 'big integer' math for the CRT. Multiply a BigInt by a ModInt, add two BigInt's. Not too difficult.

But how do we compute those constants? Some magic constant for index  $I$  &  $J$  times prime  $I$  is congruent to 1 modulo prime  $J$ . Hmmm...

That's really just a fancy way of saying: "while working modulo  $\text{Prime}[J]$ , find the inverse of  $\text{Prime}[I]$  and that's constant $[I][J]$ "

Of course, finding the multiplicative inverse is easy. We do that for the NTT anyway because we need to find the inverse of  $\text{NTTLen}$  so we can normalize our answer. (You can also use an Extended GCD to compute it.) So it's just a matter of having two loops and finding all the inverses for all combinations of  $I$  and  $J$ .

Now, what did I mean by that third point, where I said "discovering 'obvious' stuff"?

Well, with the Knuth CRT I'm describing, the primes have to be in ascending order.

For the CRT that Mikko Tomilla uses, you need to normalize the CRT. It can actually be greater than the product of your primes. So when you get done, you need to modulo it by the product of your primes. That's a little awkward, plus you need extra bits in the CRT storage to hold that overflow.

Or you can do it as you compute the CRT. You'll only need 1 bit extra.

Either way, it does take a little extra time, which is why I prefer the Knuth version.

There are other styles of CRTs, of course. Feel free to experiment.

With a simple implementation like I'm showing, it does have a growth of  $O(N^2)$  (for the CRT width, not for the NTT's length), but since we are working with CRT lengths of only a few elements, it's no big deal.



The CRT will need to be done for every set of NTT elements, so it needs to be reasonably efficient.

[Home Page](#)[SiteMap](#)[Download page](#)[Feedback form](#) [FFT Types](#)[Floating point](#)[NTT](#)

- [Galois](#)
- [Symbolic](#)

# Galois

Ahh.... The Fast Galois Transform.

It's kind of cute, although it has little to offer that a NTT can't do.

A FGT is a cross between a FFT and a NTT. The FGT uses the real / imaginary pair of a 'complex' FFT and the modular math of a NTT.

We do put the data into the FGT array differently, and we don't need to do a Complex / Real wrapper, the convolution is more difficult, and we get the data out in a different order, but all in all, you write the transform itself as expected.

I said the FGT has little to offer that a NTT can't. There are two things it has to offer.

First, a FGT can use Mersenne primes. What's good about that is the 'all bits set' nature of a Mersenne prime makes it easy to do the modulo. (The FGT can use other primes, but Mersenne primes may be easiest to work with.) You still have to do the multiplication itself, but at least the modulo is easy.

Second, with the primes used in a FGT, we can do a split-radix transform because we can now represent  $\sqrt{-1}$ . Of course, this isn't that big of a deal since the 'complex' math of a FGT results in it needing more modular multiplies than a NTT.

If you are lucky, the faster ModMul() offsets needing to do more of them, but that's not likely.

[Richard Crandall](#)'s paper on FGT's is available on the [download page](#).

Simple example code is also available on the [download page](#).

Note the difficulty we have doing the convolution.



[Home Page](#)[SiteMap](#)[Download page](#)[Feedback form](#)[▲ FFT Types](#)[Floating point](#)[NTT](#)[Galois](#)■ **Symbolic**

# Symbolic

[Schönhage-Strassen](#) | [Nussbaumer](#) | [SS vs. Nuss](#) | [Schönhage](#)

A 'symbolic' transform is one where you don't do the multiplication in the butterfly.

Sounds pretty good, doesn't it? After all, if you've read the sections on the FFT limitations, the NTT, multi-primes NTT, the CRT, etc. etc., the big problem doing a fast big multiplication is to efficiently do the multiplication in the butterfly, and do it all without anything overflowing.

Without the cost of doing the multiplications, we can make the elements as big as we need and do any size multiplication without worry.

Except, of course, for the not so small fact that reality doesn't like for us to get a 'freebie'.

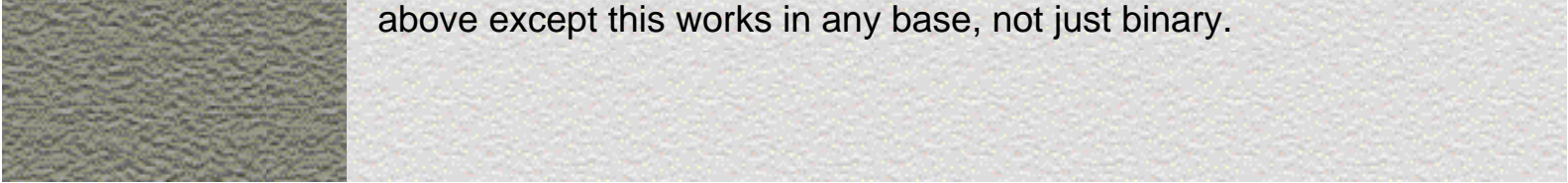
There are some costs involved. Enough costs that it may very well be better to go ahead and mess with the problems of all the other multiplication methods instead of doing this kind. The symbolic transforms actually expand the amount of data, so the convolution ends up being more expensive. All in all, it ends up that theoretically you end up with a multiplication routine that slightly lower growth than with a regular FFT or NTT.

The oldest form of symbolic transform in a multiplication is the [Schönhage-Strassen](#) multiplication. This was developed in 1970, just two years after the floating point method was discovered, and just eight years after the Karatsuba method was discovered. This method works only with binary numbers.

The other form is [Nussbaumer](#), or Nussie as I like to call it (in honor of the imaginary monster of Scotland's Loch Ness, which is often called 'Nessie'.) It's laid out a bit differently than Schönhage-Strassen, but it's been shown to 100% equivalent. The difference is that Nussie is much more flexible. First, it can work with any base, not just binary. It can be structured several different ways. The bad parts is that it takes too much memory and isn't very cache friendly. (Although with effort, both of those can be reduced.)


A third kind is, well, I guess you could call it just plain Schönhage.

Frankly, it's extremely similar to the Schönhage-Strassen I mentioned



above except this works in any base, not just binary.



[Home Page](#)[SiteMap](#)[Download page](#)[Feedback form](#) [Symbolic](#)

- **Schonhage-Strassen**  
[Nussbaumer](#)  
[SS vs. Nuss](#)  
[Schonhage](#)

# Schonhage-Strassen

[Schonhage-Strassen 2](#)

Before you read this, you should be familiar with the [NTT](#). Schonhage-Strassen is a NTT. Just a special kind. A "symbolic" NTT.

Okay. I guess you really do want to hear about SS.

(Please note. The version of Schonhage-Strassen that I'm showing is slow because it's designed that way. It was written for simplicity and as an example, rather than speed.)

Depending on your resources, you may have trouble finding decent documentation on SS, or even working examples. For most of us, the only things you are likely to find is Donald Knuth's Vol2 of "The Art of Computer Programming" (1st edition), the GMP math package (GMP) and Bruno Haible's CLN math package.

They all do the Schonhage-Strassen multiplication, but they do it differently. They all share the basics, of course. (Namely that of organizing the FFT so that the roots are easy to do. Just shifts, basically. (Actually, since it uses modular math, it's a NTT.) Plus, they all work only with binary data. Unlike a NTT or Nussbaumer.) But the actual implementations are a bit different.

First is what D. Knuth presented in the first edition of his "The Art of Computer Programming, Volume 2: Semi-Numerical algorithms". This puts the data in like normal, does a NTT multiply. There are two unique points, though. First is it uses two NTT multiplications. The normal wide one (using the special prime and root), and a tiny one holding just a few bits per element. Then a special, simple, hardwired CRT is done to combine the two into the full answer. The second part is that it pre- and post- processes the data. It scales it, much like how the 'Right Angle' FFT does. (ie: it appears to be a Discrete Weighted Transform.)

The second version is what is used in the GMP v3.1.1 math package. It does only one (the wide) NTT. It too does the scaling.

The third version is what Bruno Haible has in his CLN math package. It too only has one NTT, but it doesn't do any scaling.

Just to confuse things a bit... When I got my first version working, it did two NTTs & the CRT, but didn't do scaling.



All four possible combinations.

(Then, on top of that, D. Bernstein implies there is another version... Where as most versions are negacyclic, he says there is also a regular cyclic version. I don't have any information on that style.)

But I'm going to try and start simple. I'm going to do mostly like Knuth, but I'm going to keep it simple. Only add complexity when needed.

Let's start at the bottom and work our way up. Let's relate it to what we already know, such as a plain NTT multiplication. The differences are:

- The number of bits put into each Schonhage NTT element depends on how many total bits there are. A regular NTT uses a fixed number of bits.
- We use a 'prime' of  $2^X+1$ , where as a NTT uses  $y*2^X+1$ . The Schonhage-Strassen choice of 'prime' isn't really a prime, but in this case it doesn't matter.
- The roots are based on a power of two. This makes the 'multiply by the trig' in the NTT relatively easy to do.
- The result may be 'negative' and needs to be normalized.

There are, of course, some special conditions. For example, the NTT length has to be less than the number of bits you put into each NTT element. But, essentially, SS is like a NTT

So, my first version did it as a NTT. (As two NTT's, like Knuth describes.) To get the basic framework working, I used plain ordinary negacyclic schoolboy multiplications.

I split it up like:

```

TotalBits    = N = NumLen*2*Log2(BASE);
              n=Log2(N);
BitsPerGroup= L = Pow2(Log2(TotalBits) / 2);
l=Log2(L);
NTTLen       = K = TotalBits / L;
              k=Log2(K);
  
```



```
// NTTLen <= BitsPerGroup*2
```

```
PBits=BitsPerGroup*2;
```

```
RootExp=PBits/NTTLen; /* 2L+1-K */
```

The N, n, K, k, L, and l vars are the vars that Knuth uses to describe stuff. Lower case is the number of bits (ie:  $\text{var} = \log_2(\text{Var})$ ) Since Knuth always chooses such terrible variable names, I used things that were a little more descriptive, but left his in during development. I'm showing them here so you can relate to them if you read his text or some poorly programmed implementation from some schmuck who also thinks variables must be single letters.

TotalBits is the total number of bits in our answer.

BitsPerGroup is the number of bits I put into each element of the NTT.

NTTLen is obviously the length of the transform we'll be doing.

PBits is the number of bits each NTT/SS element will be. Our prime will be  $2^{\text{PBits}+1}$

The RootExp is the exponent of our root. It'll be  $2^{(2*\text{RootExp})}$

I choose to make NTTLen as large as possible, by doing it with that division. This is supposedly the most efficient. However, there is no requirement that you do it like that. It can be shorter. Or even a constant.

After that, the basic SS multiplication is the same as a NTT, except we compute the roots a little differently.

```
KRoot=2^RootExp;
```

```
if (Dir > 0) w=ModIPow(w,KRoot,Len/Step,ModWords);
```

```
else          w=ModIPow(w,KRoot,Len-
Len/Step,ModWords);
```

Not really all that different, but a little. The inverse just means we end up dividing, instead of multiplying. But since you can't easily divide, we arrange it as the multiplicative inverse. Like a standard NTT.

And, of course, as I said, we do that small secondary multiplication modulo NTTLen. (Remember, I did it like Knuth talks about.)

That secondary negacyclic multiplication is done modulo NTTLen and its length will also be NTTLen, so it'll always be working with fairly small data. You can do that fairly easily several different ways. A simple, naive 64 bit NTT would work fine for just about any size you need. Performance isn't critical since the length is so small. The end result will be modulo'd NTTLen.

We then use a simple hardwired CRT to combine the two.

```
Sum=Prime*((SmallMul-BigMul) % NTTLen) + BigMul;
```

See, not hard.

The final thing is that Sum might be 'negative', so we need to normalize it. (The index is 'x')

```
if (Sum >= (x+1)*2^PBits) Sum=Sum-Prime*NTTLen
```

And that's pretty much it for a \*basic\* style.

You can take a look at SS1.C in the multiplication demos collection on my [download page](#). This version is not, repeat **NOT** designed for speed. It's designed to try and be readable, so you can have some idea of what's going on. (Which kind of makes sense, since the purpose of this entire web site is to demonstrate basic multiplication, rather than just giving you finished, working high performance code that you wouldn't understand.)

Improvements are on the [next page](#)...



[Home Page](#)[SiteMap](#)[Download page](#)[Feedback form](#)[▲ Schonhage-Strassen](#)

- **Schonhage-Strassen 2**

## Schonhage-Strassen 2

If you try the example program SS1.C, you'll see it doesn't perform well. It runs about as fast as a dead dog.

Partially that's because the modular math package is so simplistic. It was not designed for performance. In particular, the modular multiplication method works a single bit at a time. Very simple and slow.

The big problem, though, is actually the Schonage-Strassen implementation itself. SS1.c was explicitly designed to show the similarities between it and a regular NTT.

Remember I previously said that SS arranges things so the 'mul by trig' in the NTT are based on powers of two, which are simple shifts...

Well, the SS1.C version doesn't take advantage of that. In fact, it can't.

So, first and foremost, we need to arrange things to get rid of explicit slow multiplications and do "fast multiplications".

And how do we do the fast multiplication? Simple, really. Our numbers are in binary and we are multiplying by a power of two.

Since  $2^{x+1}$  means we are doing negacyclic multiplication, we just do the shift and then subtract the high part (the overflow) from the shifted low part.

If our binary number was:

**abcdefgh**

and we multiplied by 8 ( $2^3$ , shift by 3) we would have

**abcdefgh000**

Doing the modulo is:

**defgh000-00000abc**

And that takes care of the forward transform. With this ModMul2Exp() style routine, we can do the forward transform with no multiplications.



The inverse transform is a little more difficult.

If you dump the trig values used in the NTT, you'll see that the inverse transform has to work with data like:

**0xffff...fff0000...00001**

That's a little more difficult to work with. What is that value? Well, actually, it's the multiplicative inverse of the power of two that we used in the forward transform. Previously we multiplied, and now we need to divide those values out.

We can deal with this two ways.

First, we can just go ahead and do the "division". Now that we know what kind of values we are dealing with (and they have a nice pattern to them), we can write a specific routine to do it. My routine `ModDiv2Exp()` does it with just a couple of shifts, and add and a subtract. Nothing really hard. But it is a little more time consuming than the forward transform. We have to do it with every single butterfly.

(Division by two isn't hard, provided you know in advance that it won't have a remainder. You multiply it by  $(\text{Prime}+1)/2$ . Which for the SS primes, works out to just two bits. A shift, an add, and a modulo (which is a shift and a subtract.) It's fairly obvious how to extend it to multiple powers of two.... This works for any Prime, not just  $2^x+1$ )

And that brings us to the second possibility. We can 'scale' the data and then both the forward and inverse transforms will be the same.

With the same nice, simple power of two multiplication. This is how Knuth describes it.

The scaling involves the value of  $2^{\text{RootExp}}$ . (The NTT root that we use is the square of this.) You just scale each element by  $(2^{\text{RootExp}})^x$ , assuming 'x' is the array index. A nice simple power of two multiplication.

The inverse scaling is a little more difficult. And it has a catch.

First, we need to swap the array ordering. It's backwards. Although element 0 is in place, the rest of them are swapped with  $\text{Len}-x$ . We just need a simple loop to do that. Or we could even leave it the way it is, and just change how we later index our answer.



Second, we need to divide it by the same scaling factor we used before. Not hard. Just a simple call to `ModDiv2Exp()`, just like what we would be doing in the NTT if we weren't using scaling. The difference is the scaling will make just one pass over the data doing the "slow" division, where as if we did it in the inverse NTT, we'd have to do it throughout the full NTT.

Either way, you'll need to do a little bit of extra effort for the modular multiplications when doing the inverse. It's not hard, but it's not quite as easy as with a single bit. It's up to you. Either way, the Schonhage-Strassen algorithm works.

The next area of improvement concerns our root. A lot of times, the low level root will be something like 2 or 4 or 16 or something small. The problem with that is to do a multiplication like that involves a lot of shifting. Computers prefer to work with full words.

The solution isn't hard.

After the area where we compute our constants, we just check to see if `RootExp` is a multiple of our word size. If it's not, we divide the `NTTLen` by 2 (or multiply `BitsPerGroup` by 2), recompute the rest of the vars, and check it again. It'll only take a couple of loops through it to get it to a nice, convenient size. Sure, it'll hurt the 'theoretical best' performance a little, but the reality is that working with full words is so much more convenient that it's well worth the effort.

It's also good to make the prime size (`PBits`) a multiple of the word size. Makes the modulo part easy to do without shifting. If you are doing numbers that are a power of two, (like the FFT likes), then you'll already have this. (And, of course, if you need to do a multiplication that isn't a power of two, Dr. David Bailey's paper (on the download page) gives a very simple way to fake it and still get good performance.)

However, even if you don't want to do this, and you end up shifting, it's not really all that hard or time consuming. Especially not when compared to doing a genuine modular multiplication.

Of course, we can also do the final normalization with the `MulDiv2Exp()` routine, instead of computing the multiplicative inverse of `NTTLen`. This way we can get rid of that function completely. Many



people prefer to do a DivBy2 on each element during the inverse transform itself, but I always prefer to wait and do it all at once.

The last area is whether to do just one NTT or two, like Knuth describes.

If you increase the width of the NTT elements to handle those few extra bits, you actually end up increasing the width by a \*LOT\*. The number of bits in each NTT element has to be a multiple of your transform length, else the 'power of two' roots don't work.

If you do the 'preferred' size of the transform vs. bits per element, you end up doubling the number of bits in each element. Obviously this is unacceptable!

However, if you limit the length of the transform, the extra bits in each NTT element will be a lot less.

You end up doing something like:

```
TotalBits    = N = NumLen*2*Log2(BASE);
              n=Log2(N);
NTTLen       = K = 16;
              k=Log2(K);
BitsPerGroup= L = TotalBits/NTTLen;
              l=Log2(L);

PBits=BitsPerGroup*2;

PBits+=(2*Log2(TotalBits));
if (PBits % NTTLen) PBits=((PBits /
NTTLen)+1)*NTTLen;
```

At this point I don't know whether that's really a good idea or not. My "gut feeling" is that it's not a good idea. That although you've made things simpler by only needing to do one transform, you've paid for it in longer run time and slightly more storage consumption. This is especially true for larger multiplications, where the extra storage and computation of the short single transform is significantly greater than a more balanced approach.

It might depend on the implementation, though.




Speaking of which.... Included in the multiplication demos on the [download page](#) is SS2.c It's a bit improved over SS1. It's gotten rid of the multiplies and now does shifts and a few other improvements.

**HOWEVER**, it's still not a high performance program. The modular math package I'm using is just not very efficient. It was written for simplicity and to be generic.

Also, a good Schönhage-Strassen implementation should do the convolution much, **MUCH** more efficiently. I'm just calling the standard (slow) ModMul routine, but you should actually recurse until your Schönhage-Strassen multiply routine is working with numbers small enough it can multiply directly. (Like how Nussbaumer and Karatsuba both do.)

But, as I said, the math package I'm using isn't written very efficiently.

In spite of that, though, all the basics for a fast Schönhage-Strassen is there. And it's public domain.

[Home Page](#)[SiteMap](#)[Download page](#)[Feedback form](#) [Symbolic](#)[Schonhage-  
Strassen](#)

- [Nussbaumer  
SS vs. Nuss  
Schonhage](#)

# Nussbaumer

[Example Nuss](#)

Nussbaumer is a cool algorithm. Not very useful but interesting none the less.

Nussbaumer convolution was discovered by Henri J. Nussbaumer in 1980. A description of it can be found in Donald Knuth's "The Art of Computer Programming", volume 2, 3rd edition, section 4.6.4, exercise 59. (Although as usual, Knuth makes it sound much much more difficult than it really is. He's a decent mathematician, but he's not much of a programmer. He's never learned the art of writing, which involves making things readable and understandable. An excellent example is that of doing a FFT. Rather than describe it like most people, he'll go into nit-picking details of one specific way to do it, totally obscuring the details. By the time you get finished reading it, you may have forgotten he was describing a FFT.)

The 'trick' to Nussbaumer is that the butterfly multiplications are done, except that it ends up being a shift. To do that, the data is pre-processed. To put it simply, it is zero padded so its length doubles.

This is one of its problems because that doubles the amount of memory required.

Another aspect of Nussbaumer is that it's a "negacyclic" multiplication. What that means is that it inherent gives an answer where the high half has been subtracted from the low half. That's one of the reasons Nussie needs to zero pad its own data. (In addition to the zero padding you've already done.)

Nussbaumer is composed of two parts.

The first part is the cyclic<->negacyclic wrapper. In other words, with some clever math it converts a negacyclic multiplication into a regular cyclic one. This can be ignored as long as you are simply willing to accept the extra memory involved in calling the main nussbaumer routine directly. And the slightly larger growth rate. This is totally separate from Nussbaumer. You can use it or Nussbaumer's negacyclic part by itself.

The second part is the main nussbaumer routine. This does the processing of the input data, does the symbolic forward transforms on the data, does the convolution, does the inverse symbolic transform,



and then post-processes the data. It's a full multiply routine all by itself.

Understanding Nussbaumer's operation is not that hard, actually.

Although it is a little complicated, the basics are pretty much the same as most FFT multiplications. It's just the implementation that's a little odd.

As far as the main Nussie convolution routine is concerned, it is a full multiplication routine all by itself.

- It takes the data and zero pads it.
- It rearranges the data into a square matrix (like the 2/4/6 step does.) Actually this step is *optional*. All it does is make the rest of it slightly more efficient.
- It does the symbolic forward transform. The transform is done almost as if you were doing a bunch of parallel transforms. (Unlike a 2/4/6 step's matrix where transforms are done both horizontally and vertically, Nussie does just one direction.) The catch is that each butterfly contains a "twist". That's the equivalent to the multiplication in a normal FFT butterfly. The difference is that the 'twist' shares data between the 'parallel' transforms. So you can't really do parallel transforms. It can be done in vector, though.
- For each row, it does 'something' to get the negacyclic product. I say 'something' because how that is obtained doesn't matter to the routine. It's generally done as a recursive call to Nussie itself (stopping when the length gets short enough to do explicitly.) That's how the official algorithm does it, and how it gets the operation count it has. However, as far as the routine itself is concerned, it doesn't matter how. You could use any method, such as a 'schoolboy', 'Karatsuba', or even a regular FFT mul. (In fact, it's actually a good idea to use some other method for the smaller sizes. That removes a lot of overhead.)
- You do the inverse symbolic transform.
- You combine the separate parts into a single answer that gets returned as the negacyclic product.



Of course, Nussie is very versatile.


- It doesn't care how the negacyclic sub-products are obtained.
- The matrix can be done either vertically or horizontally, with the twists being done horizontally or vertically.
- The transforms can be done several different ways. Regular iterative, recursive, or even as a 2/4/6 step. (Although the 2/4/6 step is more complicated this way, and it doesn't have its normal benefits of being cache friendly.)

There are a few additional points you need to know.

- The data type must be signed.
- The output data type needs to be twice as big to hold the product pyramid. That's to be expected of course, but worth mentioning.
- The only explicit multiplications are the convolution itself. The transforms shift the data.
- Because of its recursive nature and each level increasing the size of the data, in a real sense Nussie spreads the data out. What I mean is, a regular FFT makes multiple passes over the same data. Nussie makes just a few passes over that data and then spreads it out. All that extra spreading is why Nussie still requires  $O(N \cdot \log_2(N))$  multiplications, even though its symbolic FFTs don't require any multiplications. All those multiplications come from the convolution working with larger data.
- Nussie is not very cache friendly. Normally a 2/4/6 step matrix is extremely cache friendly but Nussie's isn't quite like that. And if you did the symbolic transforms as a 2/4/6 step, you would still have to do the long vectors. So there would be little benefit.
- Nussie can use recursive transforms. That helps some, but not really enough. (The best recursive transforms seem to be ones that actually do their trig scrambled, rather than no scrambling or scrambling the data. (Like I discussed in the DiT section.))



- If you use Nussie solely as a way to break up a big multiplication into a series of smaller ones, the performance isn't too bad. About twice what a good FFT/NTT method would cost. That's because Nussie doubles the length, of course. Of course, those symbolic transforms are still a problem, so it doesn't make a good disk transform.

[Home Page](#)[SiteMap](#)[Download page](#)[Feedback form](#) [Symbolic](#)[Schonhage-](#)[Strassen](#)[Nussbaumer](#)■ [SS vs. Nuss](#)[Schonhage](#)

# SS vs. Nuss

You are probably wondering which method is best....

Well... They both take about the same amount of storage. Twice what a multi-prime NTT takes, but the same as a single 62 bit prime NTT. In other words, About 4 times the number of digits.

Nussbaumer works in any base, where as Schonhage-Strassen can only work in binary. So if you need deciaml, Nussbaumer is the only choice.

I'd have to say that Schonhage-Strassen is a little easier to implement. They are both about the same level, but SS works with a standard NTT styel, which we are already familiar with. But, Nussie works with fixed width data which is easier.

Efficiency... Well, theoretically, they both have the same level complexity. They both perform the same. They have supposedly been shown to be 100% equivelent.

Practically speaking though, I'd have to say that SS would be a little easier to implement efficiently. That's because Nussie likes to do a lot of data copying and to get around that requires some clever code.

Cache efficiency is another question. That's a fairly complex area. SS is a standard NTT. We can do standard things, like recursive or 2/4/6 step, etc.

Nussie requires a cache inefficient data copy / transposition at the beginning of the transform. You can work around that if you want, but it's going to take some effort.

Nussie is normally done as a standard NumRec style transform. In other words slow. You can make it into a recursive transform, which drastically improves things. But you can't take it the next step and do a 2/4/6 step. Well, actually you can, but chunks will be larger which hurts cache efficiency. (Where as a 2/4/6 step works with 2-D data, since Nussie is already 2 dimensional, you actually end up with a 3-D cube.) You *\*might\** (and that is a big maybe!) be able to improve on that a bit, but I'm not sure. However, if you do, you will end up with some fairly complicated code. Since the data elements would be large, I don't think that's a major problem, but it would definetly be




difficult to code. Assuming it actually works like I think.

Because of the cache efficiency problem, I'm going to have to say that Schonhage-Strassen is the more efficient algorithm. I don't like it, because I normally work in decimal rather than binary, so I'd prefer Nussie to be best, but it's not.

You can generally just avoid this whole area though, and do a 62 bit prime, Montgomery modulo based NTT. That's a WHOLE lot easier to implement. Theoretically the growth isn't quite as good, but it's a lot easier to implement efficiently.

(Of course, since I wrote that, I realized that Schonhage can be done in any base, so some of the points aren't relevant.)

[Home Page](#)[SiteMap](#)[Download page](#)[Feedback form](#) [Symbolic](#)[Schonhage-](#)[Strassen](#)[Nussbaumer](#)[SS vs. Nuss](#)• [Schonhage](#)

# Schonhage

I'm stupid.

That's the only way I can describe why it took me so long to do this.

Back when I implemented Schonhage-Strassen I considered the possibility that it could be done in other bases. But I dismissed the idea because I figured that if it worked, then somebody else would have already done it. And 'everybody' always said that Schonhage-Strassen is only for binary numbers.

I understood how it was done. It's relation to a regular NTT is quite obvious. But Knuth (and others) always talked about it only in binary.

But I didn't do it. Instead, I kept going back to Nussbaumer, figuring it was the only one that could work in any base. I even started doing some pretty complicated coding just to try and make Nussie more cache friendly so it could actually be used.

All the while, in the back of my mind, I kept thinking that Schonhage-Strassen seemed like it should work in any base. Maybe Knuth just did it in binary because that's what he was interested in.

Knuth is a very "low level" person and he often gets lost in the minute details, obscuring the 'why' with the 'what'. (Reminds me of a story I heard. In the early days of computers, At a conference Alan Turing (I think) got up and started to walk through a simple algorithm on the chalk board, just as a prelude. Nobody in the audience could understand what he was doing though. Finally it occurred to somebody that what Turing was doing was multiplying two numbers. What had confused them was that for some reason he just happened to be doing it backwards. He had gotten so deep into the minute details of what he was doing that he totally confused everybody. He just implicitly assumed that everybody did things his way and that something as 'minor' as the number format was irrelevant.)

But still, I kept thinking.... As long as the relation  $\text{Root}^{\text{NTTLen}} == -1 \pmod{\text{Prime}}$  then it should work.

Well, finally I coded it. And guess what???! It works in any base!

Afterwards, I did some further checking and noticed a couple of lines



in Bernstein's paper. He gives Schönhage:  $R[x]/(x^{(mn)+1}) \rightarrow (R[x][y]/(y^{n+1}))/x^m - y$ . Notice the base of the exponents.... For the Schönhage-Strassen he gives bases of 2 and a slightly different formula. I certainly don't pretend to fully understand Bernstein's paper, but it's enough to say that this version does indeed work and isn't my imagination.

There are some differences, of course.

Instead of "bits", you now work in "digits". A single "digit" can be any base you want.

Instead of setting 'bits', you now set a digit to one.

Instead of binary shifting a number, you now shift by 'base' digits.

The multiplicative inverse (for the NTT normalization) is a little more difficult. Previously we just raised a number to a certain power. That only works when the modulus is a prime, though. We now have to use an extended GCD. Those aren't too hard to code.

**Or** we can determine the multiplicative inverse by the rule that multiplying by  $(\text{Prime}+1)/2$  is the same as dividing by two. Since this will only be  $\log_2(\text{NTTLen})$  many, we can pretty easily come up with a way to do that special multiply.

**Or** we can avoid the multiplicative inverse entirely by dividing by 2 at each butterfly during the inverse NTT. For a base 10 situation, that'd be multiplying by 500...0001 which isn't too hard. Many people do this kind of normalization directly in the FFT & NTT. I've always preferred to wait, but for this case, it's simple enough you could do it without too much effort.

The roots and such are still just single bits (in your base). You just have to make sure your low level modular math works in that base and things work out.

The inverse roots during the NTT are still done the same. Before, they were 'all bits set' on the high part, followed by zeros, followed by '1'.

Well, now they are "base-1" in the high part. The basic math still works the same.

The scaling (if desired) still works like normal.

The mini-CRT (when we release our carries) still works just fine.

You still don't need the "normaliztion" in the CRT, which checks if the number is 'negative'. Knuth has it, but I don't think it's actually needed.

The 'any base' version is pretty much the same as the regular binary version.

One difference, though, is that since we are now working in 'digits' instead of 'bits', the length and width of the NTT will be different. That will effect it's "theoretical op count", so you might want to do adjust the sizes.

I don't have an example program for this. The 'proof of concept' code I did was just an ugly hack of the previous SS1.C program already in my demonstration collection.



[Home Page](#)[SiteMap](#)[Download page](#)[Feedback form](#) [Transforms](#)[FFT Limitations](#)[FFT Types](#)

- [Cyclic vs. NegaCyclic](#)

[FFT Styles](#)[Background](#)

# Cyclic vs. NegaCyclic

While we are on the general subject of 'transform' style multiplication, I need to talk to you about cyclic versus negacyclic. And why we have to zero pad.

Cyclic means we are multiplying modulo  $\text{BASE}^n - 1$ .

Negacyclic means we are multiplying modulo  $\text{BASE}^n + 1$ .

Alright, I'm sure that doesn't mean anything to you. So I'll give you an example.

Let's say we were squaring "87654321". The full answer is 7683279989971041 of course.

Let's say our 'base' was  $10^8$ . The same size as our number.

A normal multiply would be modulo  $10^8$  and give the answer 89971041.

A cyclic multiply would be modulo  $10^8 - 1$  and give the answer 66803841.

A negacyclic multiply would be modulo  $10^8 + 1$  and give the answer 13138242

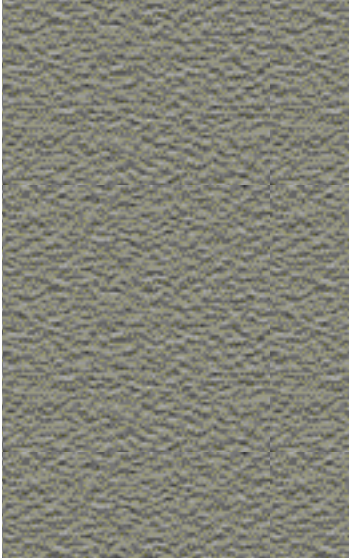
Obviously this is why we have to zero padd our number. That way we are multiplying with a base of  $10^{16}$  instead.

Doing a cyclic multiply without zero padding is as if we had done the full zero padded multiply but then added the low half to the high half. (That's how you do a base-1 modulo.)

$89971041 + 76832799 = 166803840 = (\text{wrap carry}) 66803841$

Doing a negacyclic multiply without zero padding is as if we had done the full zero padded multiply but then subtracted the high half from the low half. (That's how you do a base+1 modulo.)  $89971041 - 76832799 = 13138242$

So why do you need to know this?



Well, you don't normally. But, I thought you might like to know why we have to zero pad, and what kind of an answer you would get if you didn't.

Plus, there are occasionally uses for doing a modulo already built in.  
(If you want to do more modulo's, you could do a DWT fft, like they do in the Mersenne prime search and as described by Colin Percival.)



[Home Page](#)[SiteMap](#)[Download page](#)[Feedback form](#) [Transforms](#)[FFT Limitations](#)[FFT Types](#)[Cyclic vs.](#)[NegaCyclic](#)• [FFT Styles](#)[Background](#)

# FFT Styles

[DiF](#) | [DiT](#) | [Radix](#) | [Split Radix](#) | [Recursive](#) | [2/4/6 Step](#) | [Vector](#) | [Disk](#) | [More](#)

There are a surprising number of ways to do a Fourier style transform. Which is pretty fortunate since some are better suited to some uses than others.

They all have several common features.

The first is that all of them will be either a [Decimation in Frequency](#) (DiF) style or a [Decimation in Time](#) (DiT) style. (The names come from their original use in spectral analysis. For doing multiplications the names have no meaning, so the abbreviations DiF and DiT seem a bit more appropriate.)

The DiF style does its scrambling after the transform, while a DiT does it before the transform. (For multiplication, we often skip the scrambling, using a DiF followed by the convolution with the scrambled data, followed by an inverse DiT which unscrambles the data.)

Another common feature is how they do the scrambling. The FFT structure requires a certain amount of data movement. Some styles leave the data alone and scramble the trig powers. Most scramble the data and leave the math alone.

A third common feature is scaling. For a multiplication, after you've done the inverse transform, you need to scale the result. To divide the FFT elements by the length of the transform. You can do it directly in the FFT as part of the butterfly. You just divide by two at every butterfly. I've seen several that do that. Others do it after the transform and before returning. However, many don't do that. They just leave the data alone and let you scale the answer when use the final output (which for multiplication is releasing our carries.) It's easier that way.

No matter what style FFT we are using, it will always have those three points.

There are a number of transform styles we can use with those common features.

Most transforms you'll see are iterative. They are plain. They are



simple. These are sometimes call "Numerical Recipes" style because of their simpleness and their poor performance on monder computers.

Most FFTs break the data into two parts (left & right for DiF and even & odd for DiT) and then operate on the halves. These are called "radix-2" style. Other [radix](#) transforms are possible. Higher radix transforms do save math, but they are more complicated and the theoretical benefits are usually lost due to the extra overhead.

The '[Split Radix](#)' style is a cross between a Radix-2 and a Radix-4 transform. It's not extremely common, but does show up some times with floating point transforms because the variables just happen to fit into the registers on the x87 FPU.

Most FFTs are written in a normal 'iterative' style. However, due to the speed of the memory in modern computers, this can be a bit slow.

This style isn't very cache friendly. One way to improve the cache effectiveness is to use a [recursive](#) transform. This pretty quickly breaks the data into chunks small enough to fit into the much faster cache memory.

Another way of breaking data into chunks is called a [2, 4, or 6 step](#) transform. Where as most transform styles break data into just a few chunks, these styles usually break the data into nearly square root number of chunks. A transform of 65,536 would be broken down to 256 seperate transforms of 256 length. This is as fast as you can break the data.

The above methods break the data into chunks small enough to fit into the caches, but another aspect of modern processors is they like to have their pipelines full. This can be difficult to do with older code that works in 'scalar' mode (ie: on a single element at a time.) It's more efficient to do things in [vectors](#). In other words, long seuqences of operations. Most transforms can be re-arranged into vector form.

Nearly all transforms work best when their length is some power of two. There are some transforms that work with [non-powers of two](#) lengths, but they aren't very common and may not even run as fast due to their greater complexity. If all you are wanting to do is multiply numbers, there are better ways.

Although you can use about any style you want for an 'in memory' transform, if you want to do a [disk transform](#) you have to take special



care to reduce the amount of disk head movement. An old style transform would thrash things so badly that the constant disk head movement sound would 'drive you crazy'. It could also actually wear out your drive if you did a big enough transform. There could be many trillions of movements and drives just aren't designed for that kind of use in short period of time.

There is a lot of lore about what kind of transforms are efficient and which aren't. A lot of that lore is plain wrong. Much of it was developed back in the 60's and 70's on mainframe computers.

Today's desktop computers behave very differently than those old monsters. Even a cheap desktop has far greater usable performance than the supercomputers of the days of old. Such as the Cray-1 and Cray-2.

In these pages, I am trying to give you good current advice, but you do need to remember that there are no absolutes. Something that works well on your system may not work well on mine.

A lot of the example code is just that... **EXAMPLE** code. I'm showing you the basic algorithms and it's up to you to code them efficiently.

Also, I'm using regular 'complex' data type for the FFT examples. As the [FFT Types](#) section shows, there are other data types.

# DiF

[Home Page](#)[SiteMap](#)[Download  
page](#)[Feedback  
form](#)[▲ FFT Styles](#)• **DiF**[DiT](#)[Radix](#)[Split](#)[Radix](#)[Recursive](#)[2/4/6](#)[Step](#)[Vector](#)[Disk](#)[More](#)

No matter what other style the transform is in, it will be either a DiF (Decimation in Frequency) or a [DiT](#) (Decimation in Time). (The terms are a little confusing, since they are used in spectral analysis, but since we are doing multiplication it doesn't make sense. Just call them DiF and DiT.)

The reason these two styles are so pervasive is because of the way a regular transform is converted into a Fast transform.

At it's most basic level, a FFT is a recursive algorithm. You've only got two choices. You can do the math and then do the sub-transforms, or you can do the sub-transforms and then do the math.

A DiF does the 'butterflies' (ie: the math) and then splits the data into Left & Right halves and calls itself and then recombines the data into even & odd indexing.

By comparison, a DiT splits the data into Even & Odd parts and then calls itself and then puts the data into left & right parts.

Frankly, the one thing that makes a FFT (either DiF or DiT) awkward is having to do the "even & odd" scrambling.

In simple terms, (using C++ for its built in complex math), it looks like:

```
void RFFT_F(Cmplx *a,int n, int Dir, Cmplx *ao)
/* Recursive decimation in frequency */
{Cmplx b[MAXSIZE/2],c[MAXSIZE/2];
  int k;
  Cmplx e=exp(Cmplx(0.0,(M_PI*Dir*2.0)/(n)));

  if (n==1) {*ao=*a;return;}

  n/=2;
  for (k=0;k<n;k++)
    {Cmplx x,y;
      x=a[k];y=a[k+n];
      a[k]  = x+y;
      a[k+n]=(x-y)*pow(e,k);
    }

  RFFT_F(a,  n,Dir,b);
  RFFT_F(a+n,n,Dir,c);

  for (k=0;k<n;k++)
```



```

    {
        ao[k*2]  = b[k];
        ao[k*2+1]= c[k];
    }
}

```

This is the most basic of DiF transform. This is a direct coding of the basic FFT formula. As you can see, though, it is pretty inefficient. There is a lot of data copying and we have all that recursion to deal with.

What we can do though, is get rid of the scrambling. Then we are left with:

```

void RFFT_F(Cmplx *a,int n, int Dir)
/* Recursive decimation in frequency */
{int k;
  Cmplx e=exp(Cmplx(0.0,(M_PI*Dir*2.0)/(n)));

  n/=2;
  for (k=0;k<n;k++)
    {Cmplx x,y;
      x=a[k];y=a[k+n];
      a[k]  = x+y;
      a[k+n]=(x-y)*pow(e,k);
    }

  if (n>1) {RFFT_F(a,  n,Dir);RFFT_F(a+n,n,Dir);}
}

```

Of course, getting rid of the scrambling means it doesn't give the right answer!

We've got to do the scrambling somewhere. Fortunately, that's pretty easily solved. We just do the scrambling after we get completely done with the transform. Because of the structure of the transform it all works out okay.

The scrambling is a bit difficult to follow. Instead, I'll just say that it ends up that if you go through the array and swap an element with its bit reversal index counterpart, it works. In other words, in binary, if the index is **000110** you would swap it with **011000**.

```

void Scramble(Cmplx *data, int Len)
{int Index,xednI,k;Cmplx temp;
xednI=0;
for (Index=0;Index < Len;Index++)
{
  if (xednI > Index)
  {

```

```

        temp=data[xednI];
        data[xednI]=data[Index];
        data[Index]=temp;
    }
    k=Len/2;
    while ((k <= xednI) && (k >=1))
        {xednI-=k;k/=2;} /* bit reversal */
    xednI+=k;
}
}

```

[The bit reversal part is a little hard to follow. To put it simply, it's addition. Where as normal addition adds one to the lowest bit and the carries ripple upwards (like the normal incrementing of the variable 'Index'), this is an add to the upper bits and the carries ripple downward.]

And, of course, most people will just put the call to `scramble()` after they've done the transform. Since we are doing multiplications, if we use a DiF followed by a DiT, we can skip the scrambling completely. Saves some time.

Of course, generally you don't want to see recursive transforms like this. Instead, you'll see an iterative one. There is less overhead because we don't need to do all that recursion.

There are a couple of different styles.

```

void FFT_F(Cmplx *data, int Len, int Dir)
/* non-recursive decimation in frequency */
/* Doing the inner chunks as a chunk, */
/* rather than by common power */
{int j,k,r,step,halfstep;
 Cmplx w;
 int power;

step=Len;
while (step)
{
    halfstep=step/2;
    w=exp(Cmplx(0.0,(M_PI*Dir*(2.0))/step));

    for (r=0;r<Len;r+=step)
    {
        for (j=0;j<halfstep;j++)
            {int i1,i2;Cmplx u,v;
             i1=r+j;i2=i1+halfstep;

```



```

        u=data[i1];v=data[i2];
        data[i1]= u+v;
        data[i2]=(u-v)*pow(w,j);
        Nth=Nth*w;
    }
}
step=halfstep;
}
Scramble(data,Len);
}

```

This style is uncommon. It's pretty much a direct conversion of the recursive form to an iterative form.

The recursive form does each subchunk completely, then goes on to the next. So does this one. It just starts at the beginning of the array and sequentially does each butterfly until it's done the entire data array. It's not extremely efficient though, because it does one trig operation for each data point.

Instead, you'll generally see something like this:

```

void FFT_F(Cmplx *data, int Len, int Dir)
/* non-recursive decimation in frequency */
{int j,k,r,step,halfstep;
  Cmplx w,Nth;
  int power;

  step=Len;
  while (step)
  {
    halfstep=step/2;

    Nth=Cmplx(1.0,0.0);
    w=exp(Cmplx(0.0,(M_PI*Dir*(2.0))/step));
    for (j=0;j<halfstep;j++)
    {
      for (r=0;r<Len;r+=step)
      {int i1,i2;Cmplx u,v;
        i1=r+j;i2=i1+halfstep;
        u=data[i1];v=data[i2];
        data[i1]= u+v;
        data[i2]=(u-v)*Nth;
      }
      Nth=Nth*w;
    }
  }
}

```

```

        step=halfstep;
    }
    Scramble(data,Len);
}

```

As you can see, this is a little different.

The first difference is we are doing the trig powers differently. Before I was just using the `pow()` function to raise the trig to the needed power. It's obviously inefficient, but it was clearer and I felt that was important.

The second difference is that we are no longer doing each chunk completely. Instead, we are doing the N'th butterfly of each chunk as a group. This drastically reduces the number of math operations we need to compute the same trig values over and over and over and...

This last change is mostly just a matter of swapping the two inner loops, so they are obviously equivalent.

There is one more kind you need to know about. It's rather uncommon, but it's worth knowing about.

I really don't know what this style is called. It has the same iterative style as the first iterative one I showed you above, but it does the butterfly like a DiT. It requires a `BitReversal()` function in the FFT and a regular data scrambling afterwards. It requires less trig than the other sequential iterative one, but more than the regular iterative one.

It's sequential and the inner loop works with just a single trig value. That can be useful when random accesses are expensive and trig generation is expensive.

Although I've seen a few older transforms like this, there isn't that much use for it. The other forms are better. However, while working with the [Nussbaumer convoltuion](#), I did actually encounter a case where a recursive form of this type was more efficient and convenient than the other styles.

Frankly, there is very little use for a transform of this style.

```

int
BitRev(int Mask, int Val)
/* Bit reversal */
{
    int R = 0;

    if (Val==0) return 0;

```



```

do
{
    R *= 2; R += (Val & 1);
    Mask /= 2;
    Val /= 2;
}
while (Mask > 1);

return R;
}

void FFT_F(Cmplx *data, int Len, int Dir)
/* Simple iterative Decimation in Frequency */
/* Doing the inner chunks as a chunk, */
/* rather than by common power */
/* Does it with bitrev of the trig. */
{int ndx, halfstep, Step, r, BR, j;

for (Step = Len; Step > 1; Step /= 2)
{
    halfstep = Step / 2;
    for (r = 0; r < Len; r += Step)
        {Cmplx w;
         BR = BitRev(Len, r / halfstep);
//          w=exp(Cmplx(0.0, (M_PI*Dir*(2.0))/Len)*BR); //
or....

        w=Cmplx(cos(2.0*M_PI/Len*BR), Dir*sin(2.0*M_PI/Len*BR));
        for (j=0; j<halfstep; j++)
            {Cmplx DL, DR;
             ndx=r+j;
             DL=data[ndx]; DR=data[ndx+halfstep]*w;
             data[ndx]=DL+DR;
             data[ndx+halfstep]=DL-DR;
            }
        }
    }
}

Scramble(data, Len);
}

```





# DiT

[Home Page](#)[SiteMap](#)[Download  
page](#)[Feedback  
form](#)[FFT Styles](#)[DiF](#)[• DiT](#)[Radix](#)[Split](#)[Radix](#)[Recursive](#)[2/4/6](#)[Step](#)[Vector](#)[Disk](#)[More](#)

The text of the DiF section applies here, so I'll skip all that and get right to the code.

The basic recursive Decimation in Time transform is:

```
void RFFT_T(Cmplx *a,int n, int Dir, Cmplx *ao)
/* Recursive decimation in time */
{Cmplx
Even[MAXSIZE/2],Odd[MAXSIZE/2],b[MAXSIZE/2],c[MAXSIZE/2];
int k,x;
Cmplx e=exp(Cmplx(0.0,(M_PI*Dir*2.0)/(n)));

if (n==1) {*ao=*a;return;}

n/=2;
for (x=0;x<n;x++)
{
    Even[x]=a[x*2];
    Odd[x] =a[x*2+1];
}

RFFT_T(Even,n,Dir,b);
RFFT_T(Odd, n,Dir,c);
for (k=0;k<n;k++)
{
    ao[k]  = b[k]+c[k]*pow(e,k);
    ao[k+n]= b[k]-c[k]*pow(e,k);
}
}
```

Note the seperation into Even & Odd comes before the recursive calls and before the butterflies. Also note the DiT multiply is being done on a single element, where as with the DiF, it happens to the  $(b[k]-c[k])$  part.

An example of a recursive DiT without the scrambling is:

```
void RFFT_T(Cmplx *a,int n, int Dir)
/* Recursive decimation in time */
{int k;
Cmplx p=1.0;
Cmplx e=exp(Cmplx(0.0,(M_PI*Dir*2.0)/(n)));
```

```

n/=2;
if (n>1) RFFT_T(a, n,Dir);
if (n>1) RFFT_T(a+n,n,Dir);
for (k=0;k<n;k++)
{
    Cmplx b,c;
    b=a[k];c=a[k+n]*p;
    a[k] = b+c;
    a[k+n]= b-c;
    p*=e;
}
}

```

An example of the 'sequential iterative' version is:

```

void FFT_T(Cmplx *data, int Len, int Dir)
/* non-recursive decimation in time */
/* does it as sequential chunks */
{
    int j,k,step,halfstep;
    int index,index2;
    Cmplx temp,w;

    step=1;
    while (step < Len)
    {
        halfstep=step;
        step*=2;

        w=Cmplx(cos(M_PI/halfstep),Dir*sin(M_PI/halfstep));
        // w=exp(Cmplx(0.0,(M_PI*Dir*(2.0))/step));
        for (j=0;j<Len;j+=step)
        {
            for (k=0;k<halfstep;k++)
            {
                index=j+k;
                index2=index+halfstep;
                temp=data[index2]*pow(w,k);
                data[index2]= data[index]-temp;
                data[index] = data[index]+temp;
            }
        }
    }
}

```



And of course, the common iterative version:

```
void FFT_T(Cmplx *data, int Len, int Dir)
/* non-recursive decimation in time */
{int j,k,step,halfstep;
  int index,index2;
  Cmplx u,w,temp;

  step=1;
  while (step < Len)
  {
    halfstep=step;
    step*=2;

    u=Cmplx(1.0,0.0);
    w=Cmplx(cos(M_PI/halfstep),Dir*sin(M_PI/halfstep));
    for (j=0;j<halfstep;j++)
    {
      for (index=j;index<Len;index+=step)
      {
        index2=index+halfstep;
        temp=data[index2]*u;
        data[index2]= data[index]-temp;
        data[index] = data[index]+temp;
      }
      u=u*w;
    }
  }
}
```

Of course, the corresponding 'bitrev' version is:

```
void FFT_T(Cmplx *data, int Len,int Dir)
/* Simple iterative Decimation in Time */
/* Doing the inner chunks as a chunk, */
/* rather than by common power */
/* Does it with bitrev of the trig. */
{int ndx, halfstep, Step, r, BR, j;

  for (Step = 2; Step <= Len; Step *= 2)
  {
    halfstep = Step / 2;
```

```
for (r = 0; r < Len;r +=Step)
{Cmplx w;
  BR = BitRev(Len,r / halfstep);
  w=exp(Cmplx(0.0,(M_PI*Dir*(2.0))/Len)*BR); // or...
//
w=Cmplx(cos(2.0*M_PI/Len*BR),Dir*sin(2.0*M_PI/Len*BR));
  for (j=0;j<halfstep;j++)
    {Cmplx DL,DR;
      ndx=r+j;
      DL=data[ndx];DR=data[ndx+halfstep];
      data[ndx]=DL+DR;
      data[ndx+halfstep]=(DL-DR)*w;
    }
  }
}
```



[Home Page](#)[SiteMap](#)[Download page](#)[Feedback form](#) [FFT Styles](#)[DiF](#)[DiT](#)• [Radix](#)[Split Radix](#)[Recursive](#)[2/4/6 Step](#)[Vector](#)[Disk](#)[More](#)

# Radix

So far I've shown you code where the transform was always split into two parts. That's called a "Radix-2" transform for obvious reasons.

It's also possible to have other radices. Radix-4 and Radix-8 are some what available, although not really common. Higher ones are also around although they are very rare.

One of the problems with doing higher radix transforms is they require more overhead, more internal temporary working storage, and so on.

By careful optimization of the code and the math, it's theoretically quicker, but since the code is bigger and more complex, it may not run quite as well on modern processors.

Most processors (meaning x86) just don't have enough registers to be able to do this very efficiently. They end up storing the data into memory which reduces the performance.

You aren't really going to see too many code examples of higher radix transforms.

[Home Page](#)[SiteMap](#)[Download page](#)[Feedback form](#) [FFT Styles](#)[DiF](#)[DiT](#)[Radix](#)[Split Radix](#)• [Recursive](#)[2/4/6 Step](#)[Vector](#)[Disk](#)[More](#)

# Recursive

In the [DiF](#) and [DiT](#) sections I showed you that the *FAST* fourier transform algorithm was inherently recursive and I gave you some simple code as an example.

Well... the code I gave you was of course recursive. The very subject this section is about.

There isn't a whole lot left to say about it except that doing a recursive transform is a good way to break large transforms into chunks small enough to fit into cache memory.

As you are probably already aware, modern processors have many megabytes of slow main memory but only a small amount of fast cache memory. Naturally it makes sense to use the cache memory as much as possible just to reduce the amount of time you spend idly waiting for main memory to give you the data you need.

A regular iterative transform makes random accesses to data that is far far larger than what the cache can hold. (Random as far as the cache is concerned.) The odds are very good that the data is not going to be in the cache and the program is going to have to just sit there and wait until the slow main memory cache retrieves the data. It "thrashes" the cache.

A recursive transform is very good at improving the use of the cache. At improving the "cache locality" of the data. However, it does have too much overhead. All that recursion is just a lot of processor cycles wasted.

The solution is to recurse until you get to a size that fits fully into your cache, and then switch to a faster iterative approach.

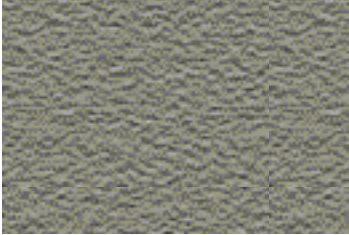
Just adding a line such as:

```
if ((sizeof(Cmplx)*Len) <= CACHE_SIZE)
    {FFT_F(Data,Len);return;} /* or FFT_T() */
```

to the recursive transform can be a significant improvement.

I could go on, but actually there is an even better method next. [David](#)





[Bailey's 2/4/6 step](#). It's a little more difficult to understand, but it's more flexible and has better cache locality.

[Home Page](#)[SiteMap](#)[Download page](#)[Feedback form](#) [FFT Styles](#)[DiF](#)[DiT](#)[Radix](#)[Split Radix](#)[Recursive](#)• [2/4/6 Step](#)[Vector](#)[Disk](#)[More](#)

## 2/4/6 Step

[Dr. David Bailey](#) is a high powered math programmer. Over the years, he's done quite a few computations, and worked with some very high powered super computers, including computing 29 million decimals of pi with the very first delivered Cray-2 back in 1986. He's also the author of the famous MPFUN fortran multiprecision math package.

(It's also worth mentioning that he has said that he's discovered flaws in nearly every computer he's ever used. From super-computers to desktop workstations. That's worth remembering if you do any significant amount of computation.)

One of the common things that is done is using a FFT. Unfortunately, he discovered that the existing transforms just weren't running as fast as they should. They were wasting very expensive super computer time.

His research led him to invent the 2/4/6/ step transform. A flexible method that significantly improved the cache locality and general performance of a FFT. Prior to him, most people were still satisfied using a fairly simple transform.

His method is rather flexible. Depending on the situation, you can do things several ways and with several steps.

His key idea is to treat the data as a matrix. Instead of a single dimension array, treat the data as a two dimensional array. You then use plain, ordinary, transforms on the columns and the rows.

The use of these small transforms improves the cache locality, which is so important in modern computers. Having to do multiple small transforms also improves the parallelization, since it allows each processor to work on independant chunks.

If you look at Dr. Bailey's papers, you'll see he's concerned with general purpose transforms. (Which makes sense, since few people do multiplication as large as we do. You'll also notice that he's a FORTRAN coder, which means his array indexing is backwards. In Fortran, the columns (first index) are sequential memory locations, where as in C, it's the rows (second index) that are sequential.)



It's interesting to note that he's humble enough to admit that even though he thought of it, it turns out that he wasn't the first. It was presented way back in 1966 by Gentleman and Sande.

(The following will assume the data is in a matrix  $X \times Y$ , with  $X$  being the width with sequential memory accesses and  $Y$  being the height with non-sequential accesses.)

The basic structure of his 4-step is:

- Perform  $Y$  number of transforms of length  $X$ .
- Scale the data
- Transpose the data into a  $Y$  by  $X$  matrix.
- Perform  $X$  number of transforms of length  $Y$

His paper goes on to discuss 2 and 6 step versions. However, they are really pretty much the same structure, except they've been modified slightly to handle limited memory or some such.

I don't happen to have any handy example code for this style because, actually, it's not very good. All that transposing of data takes time and can be a little difficult to do efficiently.

See, Dr. Bailey was concerned about generating the answer in standard FFT order. He was needing a general purpose FFT to replace existing ones.

We don't need that. Since we are doing multiplication, we can handle data that is out of order. And that lets us get rid of some scrambling and we can do a DiF/DiT pair.

Here is some example code of what I mean. This is from an old working program. It's not the most efficient, but it's all there. It's a NTT rather than a floating point FFT, but it works the same.

```
ModInt Scratch[....];

void
ColumnFFTs(ModInt Buffer[], size_t len, size_t
```

```

count,
        size_t stride,int Dir)
{int x,kount;

    for (kount=0;kount<count;kount++)
    {
        for (x=0;x<len;x++) Scratch[x]=Buffer[x*stride];
        GenericNTT(Scratch,len,Dir);
        for (x=0;x<len;x++) Buffer[x*stride]=Scratch[x];
        Buffer++;
    }
}

void Scale(ModInt *D,size_t Len,ModInt *Trig)
{int x;
    ModInt w;

    w=1;
    for (x=0;x<Len;x++)
    {ModMul(D+x,D+x,w);ModMul(w,w,Trig);}
}

void FwdTwoStep(ModInt data[], size_t NTTLen,size_t
BufLen)
{size_t k, Width=BufLen, Height=NTTLen/BufLen;
    ModInt w,Root;

    if (NTTLen <= BufLen) {FwdNTT(data,NTTLen);return;}

    w=1;
    Root=CalcRoot(NTTLen,1);

    ColumnFFTs(data,Height,Width,Width,1);

    for (k = 0; k < Height; k++)
    {
        if (k) Scale(data+k*Width,Width,w);
        ModMul(w,w,Root);
        FwdNTT(data+k*Width,Width);
/* without bit reversal scrambling */
    }
}

void RevTwoStep(ModInt data[], size_t NTTLen,size_t
BufLen)

```



```

{size_t k, Width=BufLen, Height=NTTLen/BufLen;
 ModInt w,Root;

    if (NTTLen <= BufLen) {RevNTT(data,NTTLen);return;}

    w=1;
    Root=CalcRoot(NTTLen,-1);

    for (k = 0; k < Height; k++)
    {
/* without bit reversal scrambling */
        RevNTT(data+k*Width,Width);
        if (k) Scale(data+k*Width,Width,w);
        ModMul(w,w,Root);
    }

    ColumnFFTs(data,Height,Width,Width,-1);
}

```

As I said, this isn't the most efficient structure. The columns should be vectorized, etc.

Also, it's more efficient when the Height & Width are close together. In other words, they'll be near the square root of the length of the transform.

Something like:

```

Height=Pow2(Log2(Len)/2);
Width=Len/Height;

```

In the code above, I was using a fixed size buffer (to hold the column data during the transform) and I didn't want it to overflow.

It's up to you how you want to do it. It's pretty flexible.

It's also reasonably efficient. It makes only 4 passes over the data. The columns require 1 read and 1 write and the rows require 1 read and 1 write. Everything else is small chunks that fit into the cache.

If they aren't small enough to fit into the cache, well, you can always make the above routines recursive. One level of recursion should be enough to reduce any data size into cache sized chunks.





[Home Page](#)[SiteMap](#)[Download page](#)[Feedback form](#)[▲ FFT Styles](#)[DiF](#)[DiT](#)[Radix](#)[Split Radix](#)[Recursive](#)[2/4/6 Step](#)■ [Vector](#)[Disk](#)[More](#)

# Vector

A vector transform isn't really a specific style of transform, but a way of programming one of the styles so that it runs well on modern computers.

Modern computers hate doing things one at a time. They much prefer to do a long string of similar operations on sequential memory access. In other words, a "vector". The normal FFT/NTT convolution during multiplication is a 'vector' multiply.

Doing a 'vector' transform isn't really all that hard to do. The hard part is getting into the mental attitude.

Doing a 'vector' means doing the same operation over and over. So we need to pick a style of transform where it's easy for us to do that. If you go over to the DiF or DiT section and look at the ones I show, you'll notice that the very first iterative one I show says that it works with "chunks" rather than common powers. Compare this to the normal transform, where it does one complete butterfly on one point, skips a large distance, does another butterfly, etc.

Well, "chunks" is what we need.

Each transform butterfly is just two parts... A multiply and an add/sub pair. Everything else is just a matter of indexing and varying the lengths of the chunks.

Rather than computing all those powers, we can use a small pre-computed table. The table is generated with a corresponding FFT/NTT, except instead of doing an actual transform, it just stores the trig into a table.

```
void VectorNTT_F( ModInt *Data, size_t Len )
/* A table based, vector style Decimation in
Frequency NTT */
{
    ModInt *Left, *Right, *roots;
    size_t blocks, size;
    size_t Half=Len/2;

    roots=FwdTrigTable + (MAX_TRIG_TABLE - Len);
```



```

for (size=Len/2;size >= 1;size/=2)
{
    blocks = Half/size;
    Left  = Data;
    Right = Data + size;

    while ( blocks )
    {
        VectorModButterfly( Left, Right, size );
        if (size!=1) VectorMontMul( Right, roots, size
);
        Left  += 2*size;
        Right += 2*size;
        blocks--;
    }
    roots += size;
}
}

```

If you examine it, you'll see that it relates pretty well to the sequential one on the DiF page.

This is not the most efficient, but it works. For example on the small chunk sizes, the lengths are quite small. Doing the loop overhead and the multiple calls to the vector function take time. We could hardwire a special vector routine to do that. It'd have less overhead.

You should be able to do a DiT style by yourself.

These do require the use of precomputed tables. It'd just be too inefficient to keep computing the trig over and over. (Remember, the sequential version computes a lot more trig than the more common style FFT.)

Of course, you don't want tables that are too big, either. A decent solution to that is to use a recursive vector transform. In this case, we can't pre-compute the trig tables, but we can still take advantage of the pipelines in the processor by keeping it full of vector operations.

One possible implementation is:

```

#define RVECT_SIZE 128
static ModInt PList[RVECT_SIZE];

void VectorRNTT_F(ModInt *Data,size_t Len)

```



```

/* Recursive decimation in frequency */
{size_t k,Len2;
  ModInt *Nth, *Left, *Right;
  size_t BlockLen,x,y;
  ModInt P;

  if (Len<=MAX_TRIG_TABLE)
  {VectorNTT_F(Data,Len);return;}

  Nth=&FwdNthRoot[Log2(Len)];
  Len2=Len/2;
  Left=Data;Right=Data+Len2;
  P=1;

  BlockLen=Min(RVECT_SIZE,MAX_TRIG_TABLE);
  if (BlockLen*VECTOR_RNTT <= Len2)
  {/* Only efficient (??) if we can do at least 4 */
    for (y=0;y<BlockLen;y++)
      {PList[y]=*((ModInt*)P);ModMul(P,P,Nth);}
    for (x=0;x<Len2;x+=BlockLen)
      {
        VectorModButterfly( Left, Right, BlockLen );
        VectorModMul( Right, PList,BlockLen );
        if (x+BlockLen != Len2)
          VectorModMulC( PList, P ,BlockLen );
        Left+=BlockLen;Right+=BlockLen;
      }
  }
  else
    for (k=0;k<Len2;k++)
      {
        ModAddSub(Left,Right);
        ModMul(Right,Right,P);
        Left++;Right++;
        ModMul(P,P,Nth);
      }

  VectorRNTT_F(Data,Len2);
  VectorRNTT_F(Data+Len2,Len2);
}

```

It's very much like a regular recursive transform except you do things in chunks.

Both of these code snippets are from an actual working program. So they may not be the clearest, but hopefully you'll understand the basics.

The coding itself is not really all that hard. As I said at the top, it's more a matter of getting into the right mental attitude. Once you do that, the code follows fairly easily.

Trying to convert a regular style transform into one that operates well with vectors, etc. can be difficult. But once you realize that the standard form isn't suitable and you start looking around for something else, things start making a bit more sense.



[Home Page](#)[SiteMap](#)[Download page](#)[Feedback form](#) [FFT Styles](#)[DiF](#)[DiT](#)[Radix](#)[Split Radix](#)[Recursive](#)[2/4/6 Step](#)[Vector](#)• [Disk](#)[More](#)

# Disk

Eventually your transform will be too big to fit into memory and you'll need to use a disk based transform. Writing a decent disk transform isn't easy if you don't know how. Hopefully, when we get done here, you'll know how.

The most obvious way to do it is to simply take a regular transform and depend on virtual memory.

That has two big problems. First, virtual memory is very inefficient. Virtual memory is okay for the occasional access, but all forms of virtual memory are totally unsuitable for a FFT. You'll have to manage the disk storage yourself.

The second problem has to do with the way a regular iterative transform works. It makes many passes over data that is widely spaced. You can improve things a little bit, but it still ends up needing a whole lot of disk I/O and disk head movement. That last one is particularly important because head movements are slow. And hard drives are just not made to handle trillions of head movements in a short time.

The standard kind of transform would have to make a disk head seek for every single element it read and wrote. If you tried to do a large enough transform, it's quite possible for your hard drive to wear out before it even got finished. If it didn't wear out, the universe would grow old and die before you got done.

Doing a disk transform like that should be avoided!! You probably couldn't come up with a worse kind if you tried.

So, what do we do...?

Well... That's the big question. Fortunately, after doing all this reading, there is actually a pretty simple solution. Actually two solutions.

The first is to not do any! That's right, don't do a disk transform if you don't need one. When I did my old v2.0 pi program, I used a [multi-prime NTT](#) because I needed the performance, I needed the storage savings, and because that way I didn't need to do a disk transform. That's only disk I/O efficient up to a point, but it is an option.



The second method, of course, is to do the disk transform efficiently. Don't do any more head movements than you have to. And do as little disk I/O as possible. (As long as we aren't asking for too much, let's also throw in grace, good looks, charm, and money.<grin>)

Rather than just giving you the 'solution', I'd like to walk you through a few examples. That way you'll understand why we chose the things we did.

Let's start with the simplest idea we can think of.... A plain iterative transform. How about the 'chunk' style that I show in the DiF & DiT pages? After all, I did call that one "sequential", and that sounds a lot better than the regular kind of iterative transform.

Surprisingly, that would work fairly well! Imagine that. The very first idea, the simplest idea, that we come up with would actually work.

If we assume that 'X' is how much bigger the transform is beyond our memory, it would cost us:

$$\begin{aligned}\text{Read} &= x * (\text{Log}_2(x) * 3 + 2) / 2 \\ \text{Write} &= x * (\text{Log}_2(x) * 3 + 2) / 2 \\ \text{Seeks} &= x * (\text{Log}_2(x) * 6 + 2); \end{aligned}$$

That works out to:

x=2	Read=5	Write=5	Seeks=16
x=4	Read=16	Write=16	Seeks=56
x=8	Read=44	Write=44	Seeks=160
x=16	Read=112	Write=112	Seeks=416
x=32	Read=272	Write=272	Seeks=1024
x=64	Read=640	Write=640	Seeks=2432

Remember, 'X' is how many times larger the transform is than our memory. So "x=64" means its 64 times larger than our memory. It also means that we did 640 times our memory size in disk reads. It only cost us 2,432 disk seeks, though.

It's pretty obvious there is more to worry about than just the disk head movements like I mentioned at the beginning. Sure, that's very important, but considering the speed of the disk, it'd be nice to reduce the total amount of disk I/O, too.

Can we do that. Yes.



Remember in the recursive section I mentioned how the recursive style broke the data into cache sized chunks. Well, we can do the same. And with some clever code, it will reduce the amount of disk I/O required.

Unfortunately, it won't be enough and explaining various optimizations would be difficult, so I'm not going to waste the page space discussing it further.

Rather than just recycling some 'in memory' transform, perhaps we should actually use a transform designed for disk. Surely somebody has come up with a good way to do it.

If you go over to [Numerical Recipes](#), you'll see that section 12.6 is titled "External storage or memory local FFTs". Sounds pretty much like what we want.

However, like most of the code in NumRec, appearances are deceiving. Although the method works, it was designed for 1960's era computers that used sequential mass storage devices. In other words, **TAPE** drives for storage. You know, the old, slow sequential storage device that you see in old movies. Back then, tapes did not hold much storage. An insignificant amount based on today's storage abilities.

Plus, it's not an 'in-place' transform. It copies the results to a separate storage device. That doubles the amount of storage required.

Another problem with the code is that it's designed for general FFT transforms and it returns the data in ordered format. In other words, it scrambles the output like it's supposed to. We don't need that. It doesn't hurt, but why waste the effort if we don't need it.

So how about something a little newer?

If you read David Bailey's paper on the 2/4/6 step transform, you probably noticed that he talks about doing an external storage transform. Although in his case, 'external' means a fast external ram drive (using regular memory chips), rather than a slow physical disk drive. That may seem like a minor point, but actually it's pretty significant because his "solid state disk" doesn't have any heads, and therefore there wasn't any disk head movements and it wouldn't wear out. It's also a whole lot faster than the fastest hard drive of today.



Thomas Cormen and David Nicol wrote a paper on doing transform using regular disk systems. \*\*\*\*INSERT LINK\*\*\*\* Unfortunately, their method is for parallel disk systems and it's a little cumbersome. It's not a 'drop in' replacement.

A few other minor references on the net that don't have a lot to offer.

So, if those three things aren't suitable, what's left??

Well, not a lot.

Of the three, let's take another look at Bailey's stuff. At least we can actually implement it and it's not designed for tape drives!

He shows several algorithms, but let's restate the basic 4-step: That's the simplest and a good place to start. The data is treated as a matrix of [x,y], with them being as close to  $\sqrt{\text{Len}}$  as possible in order to make the columns & rows as short as possible. (This reduces the op count and gives maximum cache locality.)

#### **FourStep**

- Perform 'x' transforms on data that is 'y' long.
- Multiply the data by some trig
- Transpose matrix from [x,y] to [y,x] indexing.
- Perform 'y' transforms on data that is 'x' long.

There are two things that makes this so unsuitable for a disk transform. The matrix transposition, and having to do so many seek operations.

The seek operations are particularly bad. It actually results in  $N^2$  number of seeks. Plus, it shows up in the transposition, too.

But the algorithm is so flexible that surely we can do something with it... After all, Bailey came up with several significant modifications, and we are smart too. Right?

Well, we can.

Doing a transposition on disk is not easy. It's not as bad as doing a



'scramble' using virtual memory, but it's not easy. And it takes a whole lot of disk head movement. It's bad enough that we flat out are going to have to solve this problem. It's not something that we can pretend that it doesn't exist. He devotes quite a bit of the paper to transposition issues. I don't want to get in too deep on this yet, so let's put it aside and go onto the next one.

The second problem was doing all the head movements to read / write the matrix. We don't have to chose our matrix dimensions to be close to  $\sqrt{\text{Len}}$ . We can actually let the width be as wide as main memory. That drops the hieght down to a very small level. And that means the number of seeks are also very small.

But what about the transposition. Well, oddly enough, by reducing the height, that also helps this part! Because the height is now so small, things are a lot easier now.

Previously, we were doing the transposition so that we could do each transform as a group of regular FFTs.

But since the height is so small now, we can just afford to load the data into memory, do the transforms, and then write it out.

In other words, instead of both transforms being in sequential order, we can now do the short height stuff as columns and the other, wider one as rows.. The height is so small we can now just load the stuff in memory

Imagine that.

What is the disk count for this, you ask? Still assuming that 'x' is how many times larger the transform is than memory....

Read= $2*x$

Write= $2*x$

Seeks= $2*x*x+2*x$

Some actual numbers are:

x=2 Read=4 Write=4 Seeks=12

x=4 Read=8 Write=8 Seeks=40

x=8 Read=16 Write=16 Seeks=144

x=16 Read=32 Write=32 Seeks=544

x=32 Read=64 Write=64 Seeks=2112

x=64 Read=128 Write=128 Seeks=8320



That's not bad. By comparison, at  $x=64$ , the first method I mentioned required 640 times the I/O, and 2432 disk seeks.

That's not bad.

Of course, the seeks is  $H \times H$ , which means it's  $N^2$ . Still though, for the size transforms were are likely to do, that's not bad overall.

By the time you get to larger sizes, you do have a problem. But by then you'll be having some major problems regardless.

To put this in perspective. If you multiplied 1 billion decimals (2048m of storage) and you devoted just 32meg of memory to the multiplication, your height would be 64 and you'd be doing just 8,320 disk seeks the whole time.

Still though, yes, I do admit that there is a point where doing the head movements will become excessive. Perhaps at 8,192 times your physical memory. (At 32m of phys mem, that would be a multiply of 128 billion ( $2^{37}$ ) decimals. Pretty big! And rather unlikely on a system with just 32m of memory available.)

There are a few additional optimizations.

Again, we can do a DiF followed by a DiT. We can do the convolution right after the row transform in the DiF, while the data is still in memory, and we can even do part of the inverse 4-step. Saves disk I/O.

Sample code is basically the same as in the 2/4/6 step section, except you change the calculation of the width & height to where the width is fixed at memory size and the height is whatever it needs to be.

And of course, you do things in chunks from the disk. That's not really all that hard. I'll leave that as an excersize for you. In the beginnin, just do it all in memory using a spare memory buffer. And pretend your physical memory is just a thousand elements. Something managable.

The hardest part is doing the column transforms. Again, you read the data in blocks. And then you either

- Copy each column data to spare memory, do the transform



and copy it back to memory

- Do a vector style transform on the mini-matrix you just loaded.

There are several ways things could be done, and it's not too hard to come up with a method that works tolerably well. (It's easier to do this than it is to get the transposition working well.)

[Home Page](#)[SiteMap](#)[Download page](#)[Feedback form](#) [FFT Styles](#)[DiF](#)[DiT](#)[Radix](#)[Split Radix](#)[Recursive](#)[2/4/6 Step](#)[Vector](#)[Disk](#)

- [More](#)

## More

For more styles, see Jorg Arndt's [FFT page](#). Be warned it has GPL code, though.

He's got a very wide variety of transforms, plus numerous links. He's building one of the largest FFT sites around. (He's doing with FFT's what I'm trying to do with multiplication.)

He also has his own 'transform' archive: FXT. A lot of code in there. A lot of quality. A lot of effort.

However, it does have a few faults.

First, it's in C++. That doesn't help portability. True, some things are easier in C++ (such as complex math), but a lot of the other features get in the way of portability.

Second, it (and HFloat) appear to be written solely for Linux, so 95% of the population can't use it! Although I've never tried compiling his stuff, several people have repeatedly told me that it flat out will not compile under Windows. Considering the Linux flavor of the code, they may be right. At the very least, it would probably be correct to say it doesn't compile easily with DJGPP or MINGW

Third, the style of his transforms is a little unusual. Both the code structure itself, and the data layout. He's one of the very few odd people who prefer not to use the standard layout for FFT data. Suffice to say, that does complicate using his code!! I can't think of even one single good reason why he deliberately did that. [disgust] It goes against tradition and on modern processors, needing to do two widely spaced accesses to get just one single piece of data is guaranteed to be slow.

Fourth, his code falls under the GNU Public License (GPL). What that means is you can't use his code unless you release your code as GPL. If you include even **ONE** line of GPL'ed code in your own public domain million line program, then your entire program has to be released as GPL. If you want to release your code as public domain, you can't do it.

Frankly, that kind of license is one that should be intently avoided. It's a legal virus. It infects your own code. It infects and spread very



much like a regularly biological virus or a computer virus. The linux fanatics angrily point to Microsoft and say that you can't even use their code. They are totally oblivious to the fact that their own behavior is even worse. They try to create open code by **FORCE**. (It may even be illegal. Nobody has ever actually tested it in court.) It's like a thief forcing you to 'donate' your wallet to him because he's pointing a gun at you. The GPL supporters say that you don't have to use GPL.

That's true. Provided you are aware of the virus like nature of the GPL code, you can make that choice. But that thief can also claim that you chose to walk down that dark street of your own free will and that he shouldn't be held accountable for stealing your wallet.

This one aspect of the GPL is the reason so many companies refuse to contribute to the GPL and refuse to have any GPL code on the premises. They are willing to use compiled programs, but many companies refuse to allow GPL'ed code anywhere near their own stuff.

And for good reason! Can you imagine what would happen if, say, Microsoft accidentally used 1 line of GPL code in Windows? That's right... They'd have to make the Windows source fully available.

I talked with Joerg about this once and he was shocked to discover this aspect of the GPL and he told me that he would immediately change his license. Well, he hasn't. So I have to assume that he enjoys this kind of behavior from supposedly open code. That he does indeed approve. (I know Bruno Haible, author of CLN, approves.)

### **AVOID USING ANY OF JORG's CODE!**

Feel free to use it as example, but avoid using it. There's a lot of decent algorithms in there, but as long as Joerg has that attitude, I simply can not endorse the use of his code. He's turned all his hard work into a "read only" library. Where you can "look but not touch".

There are other publicly available licenses that could be used. That encourage open code without attempting to steal your own. Just right off the top of my head, something like the BSD, MIT, or zlib/libpng licenses would be a good place to start. Or you could even go fully freeware, like Mikko Tomilla did with his APFloat package. There are plenty of alternatives to the GPL virus.

You can safely follow the links on his site, of course. He hasn't yet figured out any way to put a license on those. (Although at one time, he did jokingly (?) have a notice on his site saying that anyone from

Microsoft who used his links had to pay him...)

As I said, he's put a lot of work into his code, but as long as it is released under the GPL virus, don't let it anywhere near your own code.

It is worth pointing out that in spite of my distaste for his code license, I do have to recommend you read his "FXT book" on his FXT page. It has a lot of information on the FFT, FHT and NTT. It's excellent reference material. A lot of information.

You might also want to check the [FFTW](#) people. Their code is also under the GPL, so avoid using it. But you can still use the links to other sites.



