apfloat

A C++ High Performance Arbitrary Precision Arithmetic Package

Version 2.33

September 9th, 2001

Mikko Tommila
E-mail: Mikko.Tommila@iki.fi

Abstract

Apfloat is a C++ arbitrary precision arithmetic package. Multiplications are done using Fast Fourier Transforms for O(n log n) complexity. The transforms are done as Number Theoretic Transforms to avoid round-off problems. Three different moduli are used for optimal memory usage. The final result is achieved using the Chinese Remainder Theorem. The algorithms are optimized for very high precision (more than 100 000 digits). The package is written to be easily portable, but also includes assembler optimization in critical parts for various processors for maximum performance. The software is released as freeware and is free for non-commercial use.

This document and the software are located at `http://www.iki.fi/~mtommila/apfloat/`

2

Table of Contents

Legal Notice

This program (the apfloat source code and documentation) is freeware. This means that you can freely use, distribute, modify and compile it, but you can't sell it or any part of it. Basically you can do anything with it, but the program or any part of it will always be free. That is you can't charge money or other valuables or services for it.

Although you can use this program freely, it would perhaps be considered to be good manners to give the original author credit for his work, if this program is ever used for anything useful or remarkable.

The author takes no responsibility whatsoever for any damage or harm that could result from using this program. The program has been thoroughly tested, so using it should be fairly safe. However, executing it as root is perhaps not a very good idea.

Once more (a standard disclaimer):

THIS SOFTWARE IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PRODUCT IS WITH YOU. SHOULD THE PRODUCT PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

IN NO EVENT WILL MIKKO TOMMILA, THE AUTHOR OF THIS SOFTWARE, OR ANY OTHER PARTY WHO MAY HAVE REDISTRIBUTED THE PRODUCT AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PRODUCT (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PRODUCT TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

1. Introduction

The original idea for this program got started from the author's personal interest in calculating π to as many decimal digits as possible as fast as possible. It's difficult to imagine any (other) reasonable use for this program. Calculations like this can of course be used for example to test a computer system's reliability since a single error in one arithmetic instruction will render the rest of the calculated digits totally wrong. There could be a bug in this program also.

Use of this package has been made as simple as possible so that the user's need for special customization and knowledge of the inner structure of the program is minimized. Despite the

simplicity the program is nearly as efficient as what would be achieved with customized tricky programming.

The author is aware that there exist several other similar multiprecision packages (like [11] and [12]). This program was written because of the author's personal interest in the subject.

All comments about the program and especially bug reports should be sent by e-mail to the author (Mikko.Tommila@iki.fi).

2. Compiling the Library

First unpack the compressed source file and the appropriate makefile package for your compiler: djgpp, bcc32, vc, Linux or general UNIX gcc, or just any general C++ compiler (the makefile is for gcc, so you may want to change that). Then simply run **make lib**.

If you use a UNIX system, you may need to modify the makefile to tell the compiler for example to enable integer multiplication and division instructions or to set long ints to be 64-bit (if you use the 64-bit version). Simply add the required options to the **OPTS =** line in the makefile. On most platforms, however, you should be able to compile the code without any changes. The file readme.1st has more troubleshooting hints.

3. Using Apfloats

Using the apfloat library is simple. After compiling the library you only need the header file apfloat.h plus the compiled library. In each file you plan to use apfloats in you should always **#include "apfloat.h"**. Then simply write a program like

```
#include <iostream.h>
#include "apfloat.h"

int main(void)
{
    apfloat x = 2;

    x.prec(1000);
    cout << sqrt(x) << endl;

    return 0;
}
```

and compile it with the apfloat library (apfloat.a or apfloat.lib) you created before.

3.1 Constructors

You can construct an apfloat from an integer, a double, a character string or another apfloat. Integers have infinite precision by default (actually **0x7FFFFFFF** base units in a 32-bit address space), doubles about 16 decimal digits and strings the precision of the string length. One base unit is 10^9 or 9 decimal digits in 32-bit implementations, 19 digits in 64-bit implementations and 15 or 7 digits in the floating-point implementations (doubles or floats correspondingly). For example:

```
apfloat a = 5;          // Infinite precision
apfloat b = 5.0;        // Precision is about 16 decimals
apfloat c = "123.4567890123456789012345678901234567890";      // About 30 decimals
```

The constructors have the precision as the second optional argument. For example:

```
apfloat x = apfloat(5, 1000);          // Precision is 1000 digits
apfloat y = apfloat(1.5, 2000);        // Precision is 2000 digits
apfloat z = apfloat("123", 3000);      // Precision is 3000 digits
```

3.2 Arithmetic Operations and Functions

The standard arithmetic operations

```
+
-
*
/
+=
-=
*=
/=
++
--
```

are overloaded for the apfloat class. Also the following functions are overloaded:

```
invroot(x, n) // Inverse nth root (using Newton's iteration)
root(x, n)    // Integer nth root (inverse of invroot)
sqrt(x)       // Square root (optimized)
cbrt(x)       // Cube root (optimized)
pow(x, n)     // Integer power
floor(x)      // Floor function
ceil(x)       // Ceiling function
abs(x)        // Absolute value
modf(x, *i)   // Splits to integer and fractional parts
fmod(x, y)    // x modulo y
agm(x, y)     // Arithmetic-geometric mean
log(x)        // Natural logarithm
exp(x)        // Exponential function
pow(x, y)     // Arbitrary power x^y
sin(x)        // Sine              (included in apcplx.h)
cos(x)        // Cosine            (included in apcplx.h)
tan(x)        // Tangent           (included in apcplx.h)
asin(x)       // Inverse sine      (included in apcplx.h)
acos(x)       // Inverse cosine    (included in apcplx.h)
atan(x)       // Inverse tangent   (included in apcplx.h)
atan2(x, y)   // Angle of (x, y) on the complex plane (in apcplx.h)
sinh(x)       // Hyperbolic sine
cosh(x)       // Hyperbolic cosine
tanh(x)       // Hyperbolic tangent
asinh(x)      // Inverse hyperbolic sine
acosh(x)      // Inverse hyperbolic cosine
atanh(x)      // Inverse hyperbolic tangent
```

Division uses the invroot function.

There is a function `pi(prec)` which gives π calculated to `prec` digits.

There are also stream input and output operators, so you can for example

```
apfloat x = "3.1415926535";
cout << x;
```

This outputs the number in a floating-point style number, like

```
0.0000000031415926535e9
```

If you want a prettier output (no exponent, all the digits), there is a manipulator:

```
cout << pretty << x;
```

will output

```
3.1415926535
```

3.3 Member Functions

Apfloats have the following member functions:

```
int sign(void);
void sign(int newsign);
long exp(void);
void exp(long newexp);
size_t prec(void);
void prec(size_t newprec);
int location(void);
void location(int newlocation);
void unique(void);
void swapto(char *filename);
void swapfrom(char *filename);
```

The `sign()` function returns the sign of the number (1, 0, or –1 for positive, zero and negative numbers correspondingly). `sign(s)` sets the sign to s.

`exp()` correspondingly returns and sets the exponent. Note that the exponent can only be set in multiples of the number of digits in one base unit.

`prec()` returns and sets the precision. There is a constant named `INFINITE`, which can also be used. It's the precision integers are set to by default.

`location()` returns and sets the location of the data in the mantissa of the number. It can have one of the constant values defined in apfloat.h: `MEMORY` or `DISK`. There's no reason to use this function and moving too big numbers to memory can cause the program to abort or crash unexpectedly.

`unique()` ensures that the data of the number is a unique copy. Due to the pointer structure of the program more than one number can point to the same data. There should be no reason to ever use this function.

swapto(char *filename) "swaps" the number to the specified file. That is, the number is saved to disk and deleted from your program (the number becomes uninitialized). The function is implemented so that if the number already resides on disk, this function does very little (just appends the number's member fields to the data of the mantissa) and is very efficient. This is an useful function for saving numbers to disk for e.g. transferring them between programs. It is far more efficient than printing and inputting the number via file I/O streams.

swapfrom(char *filename) "swaps" the number from the specified file, that is loads it from a file where a number was saved previously with **swapto()**. The specified file is essentially deleted from disk. Again, if the number is very big and should by default reside on disk, this function is very fast.

Mostly you will only need the **prec()** function.

3.4 Complex Numbers

Complex arithmetic can be done with the **apcomplex** data type. The necessary declarations are in the file apcplx.h. All the apcomplex functions are compiled in the apfloat library.

Apcomplex numbers relate to apfloats just like standard C++ complex numbers relate to doubles. An apcomplex number is constructed from two apfloats: the real part and the imaginary part. For example:

```
apcomplex z = apcomplex(0, "1e1000");
```

All the mathematical functions are also overloaded for the apcomplex type, as are the stream input and output operators. Also the standard C++ complex manipulators (real, imag, conj, norm and arg) and the polar constructor are overloaded.

The real and imaginary parts of an apcomplex number can be directly accessed as the members re and im. For example:

```
z.im.prec(100);
```

The apcomplex class also has a **prec()** member function, which returns the precision of the number. The precision cannot be set this way, it must be set explicitly via the members re and im.

Note that in order to use the real trigonometric functions (sin, cos, tan and their inverses), you must include apcplx.h, since these functions are calculated via complex functions.

There are some examples of complex arithmetic in the file cplxtest.cc.

3.5 Integers

Integer arithmetic can be done with the **apint** data type. The necessary declarations are in the file apint.h. All the apint functions are compiled in the apfloat library.

Apint numbers relate to apfloats just like standard C ints relate to doubles. An apint number is an arbitrary precision integer. For example:

```
apint i = 100;
```

All the arithmetical operations are overloaded for the apint type, including the modulo `%` and `%=` operators and the shifting operators (`<<` and `>>`). Also the stream input and output operators are overloaded. Apints are always output with full precision (the `pretty` modifier is used for the output). Arithmetical operators with other arbitrary precision data types are also overloaded. Conversion from apint to apfloat should happen automatically when necessary. The precision of an arbitrary precision integer is naturally always infinite and it cannot be changed. Also the arithmetic with apints works with exact precision always. This is obviously required for integer division and modulus.

The following mathematical functions are implemented for the apint class:

```
pow(x, n)        // Integer power
abs(x)           // Absolute value
div(x, y)        // Splits to quotient and remainder, returns apdiv_t
gcd(x, y)        // Greatest common divisor
lcm(x, y)        // Least common multiple
powmod(x, y, m)  // Integer power modulo a modulus
```

There are some examples of arbitrary precision integer arithmetic in the file inttest.cc.

3.6 Rational numbers

Arbitrary precision rational arithmetic can be done with the `aprational` data type. The necessary declarations are in the file aprat.h. All the aprational functions are compiled in the apfloat library.

An aprational number is constructed from two apints: the nominator and the denominator. For example, the following code declares the rational number 2/3:

```
aprational r(2, 3);
```

All the elementary arithmetic operations are overloaded for the aprational type, as are the stream input and output operators.

The nominator and denominator of an aprational number can be directly accessed as the members nom and den. For example:

```
cout << r.nom;
```

As the members of the aprational class (the nominator and the denominator) are integers, both of them have infinite precision. This can't be changed. You can get a floating-point approximation of the rational number with the member function `approx(prec)`, which returns an apfloat with the desired precision `prec`. You can mix apint and aprational numbers in arithmetic operations, but when you are using apfloats with aprational numbers, you should

always use explicit floating-point approximations of the rational numbers with the member function `approx().`

Because rational numbers are not uniquely defined, unless the nominator and the denominator have no common factors, this arises some questions after every arithmetic operation is done. Should the nominator and denominator be reduced so that they have no common factors? As this can be quite tedious and sometimes is not necessary, there is a static member variable called `autoreduce`. By default it is set to `true`, which means that after every operation the nominator and denominator are reduced to the smallest possible numbers. If it is set to `false`, this reduction is not done and the nominator and denominator can grow unnecessarily big. This can speed up things, if it is known that the nominator and denominator will have no significantly large common factors. You can still manually reduce the rational number to the smallest possible numerator and denominator by calling the member function `reduce()`. The reduction simply first calculates the greatest common divisor of the nominator and denominator and then divides the nominator and denominator by the gcd. Because this can be highly inefficient, it is recommended to always set the `autoreduce` parameter to false if it is feasible.

The function `pow(x, n)` is overloaded for the aprational class (for the parameter `x`).

There are some examples of rational arithmetic in the file rattest.cc.

3.7 Things to Note

- When the numbers are stored on disk, the program will create temporary files in the current directory. The files are named xxxxxxxx.ap, where xxxxxxxx is a number starting from 00000000. Naturally you should have permission to write files in the current directory.
- Remember to set integers to a finite precision before doing arithmetic on them which will create an infinite decimal expansion (like `sqrt(2)` or 2/3). For example

```
apfloat x = 2;
cout << sqrt(x);
```

will exhaust virtual memory or result in a crash. Instead define the precision in the constructor:

```
apfloat x = apfloat(2, 1000);
cout << sqrt(x);
```

or afterwards, like

```
apfloat x = 2;
x.prec(1000);
cout << sqrt(x);
```

- It probably doesn't make much sense to construct high-precision apfloats from numbers with infinite binary expansions using the constructor from a double. For example

```
apfloat x = apfloat(1.3, 1000);
```

will be correct only to at most 16 digits, not 1000. This is because the number 1.3 cannot be presented exactly in base two with a finite number of bits (which is the case when you use a double). Depending on your compiler there might be an error of about 10^{-16} with any doubles (like 0.5). Instead you should use

```
apfloat x = apfloat("1.3", 1000);
```

- The compiler will probably give a lot of warnings when you compile the code. This is due to the structure of the apfloats. Since an apfloat only contains a pointer to the actual data and only pointers are exchanged in constructors and assignment operations, temporary objects will be used in suspicious constructors. For example

```
apfloat x = apfloat(2, 1000);
cout << sqrt(x) << endl;
```

will use a temporary apfloat. The first line constructs an `apfloat(2, 1000)`. On the second line it's copied to the parameter that goes to `sqrt()`. If all the data was copied a lot of time and space would be wasted. Only a link to the actual data is added and then later removed at the function return, so much time is saved. A temporary object `sqrt(x)` is created. It is then output to cout. Then the temporary object is destroyed. There is nothing wrong with this, but you'll get a warning.
- This package is designed for extreme precision. The result might have a few digits less than you'd expect (about 10) and the last few (about 10) digits in the result might be inaccurate. If you plan to use numbers with only a few hundred digits, use a program like PARI (it's free and available from `ftp://megrez.math.u-bordeaux.fr`), or a commercial program like Mathematica or Maple if possible.

3.8 Using Some Other Base than Base 10

If you want to do calculations in some other base than decimal (base 10) use the `apbase()` function. Note that you can't change the base between calculations (or you shouldn't, since it will result in a crash). That is, your code should delete all the apfloats crated so far before changing the base. Thus it's a good idea to change the base in the beginning of your program and then not change it after that. For detailed instructions refer to the file bases.txt in the package.

4. Classes and Data Structures
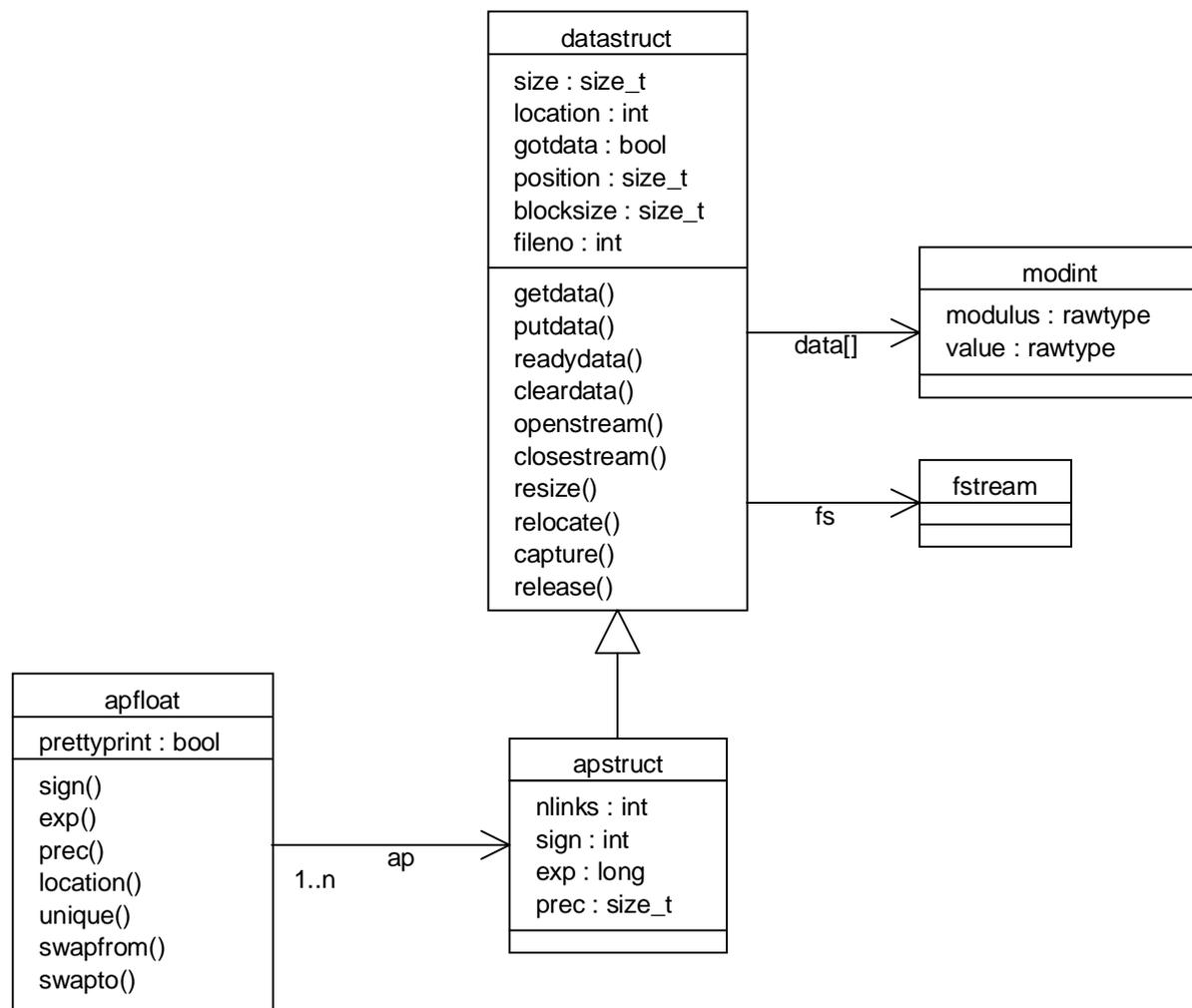
Figure 1: Class diagram

Practically all the work the program does is done on the datastruct class. The datastruct stores the mantissa of the number, that is all the significant digits. The data is accessed through the **getdata()** and **putdata()** functions. The data itself can reside either in memory or on disk.

When a function wants to use the data of the mantissa, it calls **getdata(position, size)**. It returns a pointer to the data. If the number was located in memory, it only returns the pointer to the beginning of the data plus the parameter **position**. If the number was on disk, a buffer of size **size** is allocated and the data from the appropriate position in the file is read to the buffer. Then the address of the buffer is returned. Thus the data will be accessed the same way whether it is located in memory or on disk.

When the function is done with manipulating the data, it calls **putdata()**. If the number was located in memory, putdata does nothing since the function already changed the data in the right position. If the number is located on disk, putdata writes the data to the right file position and frees the memory that was allocated for the buffer.

Correspondingly there also exist functions `readydata(position, size)` and `cleardata()`. Readydata only readies the buffer for writing data to the position (with `putdata()`), but nothing is assumed of the previous contents of the data in that position. Cleardata just deallocates the buffer that was previously allocated with `getdata()` for reading purposes only.

The datastruct class also naturally includes the data size and the location of the data.

The datastruct class includes only the raw data of the number. The apstruct class is derived from the datastruct class. It includes all additional data about the number: sign, exponent, precision and number of links to the data.

The apfloat class, the only part visible to the end user, only contains a pointer to an apstruct. This way apfloats can be used effectively just like normal floating-point numbers in C++. Every time a number is passed to a function as an argument or assigned (the = operator) to another variable a copy is made of the number. If all the data (possibly tens of megabytes) was copied every time, a huge amount of time and space would be wasted. This is why copying apfloats means only that the pointer is copied and the number of links in the data is increased by one. If the number needs to be changed (for example by changing the precision with the `prec()` member function), an original copy is first created with the `unique()` function. When an apfloat is destroyed, only a links is removed from the data. If the number of links to the data is zero, then the actual data is destroyed. All arithmetic operations always create a new (temporary) apfloat, so this method works very well (and it's completely invisible to the user).

Some arithmetic operations, like addition, subtraction and multiplication also benefit from the pointer structure. If the arguments (the pointers) to the operation are identical, the numbers are known to be equal. Especially multiplication becomes squaring, which is a lot faster.

Since multiplication of apfloats (the most intensive part of the program) is done using Number Theoretic Transforms (see appendix A), the data is stored as the modint class. Modints are integers, but the arithmetic operations are overloaded so that the arithmetic is always done modulo the global variable modint::modulus (or sometimes modulus). This makes coding the transform functions very simple and intuitive, yet very effective due to inline functions. Even assembler optimization is possible because gcc supports very flexible inline assembler statements (see section 7).

The program also has a "bigint" data type, which is basically only used for temporary calculations in the Chinese Remainder Theorem (see appendix B). Bigints are simply arrays of unsigned integers. Relatively short arithmetic (like 96-bit) is fastest using bigints, since the hardware in general directly supports them.

Sometimes really big blocks of modints need to be allocated (tens of megabytes). Most compilers or operating systems seem to handle allocation of very big memory blocks quite strangely. When the allocated block is freed, the memory is not actually freed, but somehow just marked free. If a larger block is allocated right after that, the previously allocated block cannot be re-used and almost double the necessary memory gets allocated. This will result in

either running out of memory or extreme swapping and heavily degraded performance when running at the limits of available memory (which should be the case).

Before any operations on the apfloat class are done, certain initial functions must be performed. These are done in the function `apinit()`. To automatically call this function at the beginning of the program, and the function `apdeinit()` at program exit, which cleans up the things that `apinit()` did, a dummy class is constructed in the file init.cc. This class, apfloatinit, has only one instance and it is static. The constructor for this class simply calls `apinit()` and the destructor calls `apdeinit()`. Thus at the beginning of the program, before `main()` is run, the constructor for this static apfloatinit instance is of course called. After `main()` has finished, the destructor for the instance is called. So, the `apinit()` and `apdeinit()` functions are performed automatically.

All the data in the program is allocated and deallocated dynamically during run time with the `new[]` and `delete[]` operators. However due to the memory allocation problem mentioned above a slightly different approach was implemented. At the program start, in the function `apinit()`, a memory block of maximum size (power of two or three times a power of two) is allocated. The pointer is in the global variable `workspace`. It's freed at program exit, that is when `apdeinit()` is called. Fortunately, C++ makes possible to overload the `new[]` and `delete[]` operators for the modint class so that every time a block larger than the global variable `Memorytreshold` (see section 5) is "allocated", a pointer to workspace is returned. It's never deallocated with `delete[]`. If the workspace is "allocated" twice, the program aborts with an assertion failure. This should never happen, however, since normally numbers larger than `Memorytreshold` are stored on disk. Only one (large) buffer at a time is allocated in memory for intermediate computation results. Blocks smaller than `Memorytreshold` are allocated normally with the rawtype's `new[]` and `delete[]` operators. This is why you should never change a number's `location()` unless you know exactly what you are doing.

5. Adjusting System Parameters

The file apfloat.ini (must be in the current directory) can contain some (optional) information about your system. It's highly recommended that you check the values especially if you plan to do very long calculations. It can have the following lines:

`Ramsize=41943040`
This is your computer's memory size in bytes. An estimate of the program and operating system code size is subtracted from this value to get the maximum available power-of-two or three times a power of two block size.

`CacheL1size=8192`
The processor's level-1 cache size in bytes. This has no great effect on the performance, so if you don't know it, you should probably leave it to 8192.

`CacheL2size=262144`
The processor's level-2 cache size in bytes. This has no great effect on the performance, so you should probably leave it alone unless you are an optimization freak.

`Cacheburst=32`
The cache's burst size (or a cache line size) in bytes, typically set to 32 or 16. Again this has no big effect. The cache parameters are only used in matrix transposition in the transform algorithms (see appendix D), which takes only a very small part of the total program CPU time.

`Memorytreshold=131072`
Longer data blocks than this are stored on disk by default. When the numbers are stored in memory, the program runs slightly faster. Don't set it to too high or the program will swap or run out of memory.

`Blocksize=65536`
Efficient disk I/O block size (in modints), should be $\le$ `Memorytreshold`. Since a lot of data is read from the disk in reverse order, you should probably set this to a quite large value for good performance.

`NProcessors=1`
Number of processors in a multiprocessing system. For normal, single-processor desktop computers the default value of 1 should be used. Not all versions of apfloat use this parameter. Currently multithreaded versions of the six-step Fast Number Theoretic Transform are implemented for Win32 threads and Posix threads. Also a multiprocessing program for calculating pi is included in the apfloat package.

All of the above parameters can also be specified as environment variables. The environment variables should be in uppercase (for example, `RAMSIZE`). The environment variables override the settings in apfloat.ini, if both exist on a system.

A `MAXBLOCKSIZE` environment variable can also be specified, which will override the setting calculated from `RAMSIZE`. This variable can be used to directly specify the maximum available memory block size, in modints. It should be used with caution.

6. Multiplication of Very Large Numbers

This chapter was inspired by [5].

A multiplication is essentially a convolution. For example, consider the numbers 123 and 456 presented as the sequences {1, 2, 3} and {4, 5, 6}, respectively. The linear convolution is presented in figure 2. If we want to do the calculation using a circular convolution (and we will), the sequences must be zero-padded to the length of the sum of the lengths of the operands. In this case the sequences would then be {1, 2, 3, 0, 0, 0} and {4, 5, 6, 0, 0, 0}.

```
              1    2    3
         ·    4    5    6
              6·1  6·2  6·3
                   +    +
         5·1  5·2  5·3
              +    +
    + 4·1  4·2  4·3
      4¹   13³  28²  27¹  18
      5    6    0    8    8
```

Figure 2: Convolution

From the convolution sequence carries must be calculated, since the numbers of the sequence can be larger than the base used. For example if base 10 is used and the result from the convolution is the sequence {4, 13, 28, 27, 18, 0}, it must be converted to a sequence where all the numbers are less than 10. Since each number in the sequence represents one "digit" in

the decimal expansion of the result, the 18 in the least significant position actually means that the least significant digit is 8 and 1 should be added to the next digit. It's 27, so the second least significant digit of the result is the lowest digit of (27+1) which is 8, and 2 is added to the third least significant digit etc. Basically this is the standard addition scheme, as shown in figure 2.

The convolution can be calculated very effectively using Fast Fourier Transforms. Normal complex Fourier Transforms break down due to round-off errors with transforms longer than a few million (according to [5]), so the transforms are done in the field of integers modulo a prime p where p is of the form $kN+1$ and N is the transform length. This method also has several other advantages like a simpler and more efficient memory usage. For a thorough discussion, see appendix A.

In this program three different moduli are used for the convolution and the result is acquired with the Chinese Remainder Theorem. This has the advantage of effective memory usage since the numbers don't need to be split up to smaller parts to avoid overflow. Now if the modulus is about 2^{31} and the base used is 10^9, the maximum precision would theoretically be about 10 billion base units or 90 billion decimal digits. However there are three primes less than 2^{31} of the form $kN+1$ only up to $N=2^{25}$ when N is a power of two. Actually these three moduli allow N to be $3 \cdot 2^{25}$, so the maximum transform length can be increased by 50% with a suitable transform algorithm. This corresponds to about 220 million decimal digits. If more digits are desired, a larger modulus must be used. This requires use of 64 to 128-bit arithmetic or other tricks. In the 64-bit implementation the maximum precision is about 60 billion digits and in the floating-point implementations about 790 trillion in the double version and slightly more than a million in the short version.

After multiplication can be done efficiently, division, square root and other roots can be calculated using Newton's iteration:

$$x_{k+1} = x_k + \frac{x_k(1 - a\, x_k^n)}{n}$$

Now x_k converges quadratically to $a^{-1/n}$.

The program includes optimized routines for multiplication and division when the multiplier or divisor is a "short" number, in division a number with about 9 significant digits or less and in multiplication about 450 digits (depends on your platform).

7. Performance Considerations

Although this program is optimized for optimal use of memory, you should have at least so much memory that the transforms can be done in memory. Although the program has optimized "two-pass" external memory transform algorithms (see [4] for the algorithm), disk speed is so slow that performance will be close to zero (it can be more than 30 times slower compared to the transform being done in memory). This means you should have at least about 1 free byte of memory per 1.125 decimal digits in the largest (longest) number in the calculation. Maximum available memory is used only in power of two or three times a power of two block sizes. For example if you have 32 MB of memory, you can use 24 MB of it for the data if the program and the operating system fit in the 8 MB.

Disk speed is crucial if the numbers are stored intermediately on disk. With a relatively fast CPU but a relatively slow disk the overall performance will suffer. Having an efficient disk cache is also highly recommended.

For some reason the Borland C++ and Visual C++ versions don't work as fast as the djgpp versions on Windows 95/98, although the code is virtually identical. On Windows NT the performance is about the same. The suspected reason is in how Windows 95/98 manages its memory. For maximal speed, you should get djgpp if you use MS-DOS, Windows 95 or Windows 98. It's free, very flexible and it optimizes well. Djgpp is downloadable from `http://www.delorie.com/djgpp/` and various mirror sites.

Note that the djgpp versions can't probably use all the memory you may have installed on your computer. Djgpp v1 seems to be limited to allocating 64 MB and djgpp v2 may be limited to 256 MB. If you have more memory than this and want to utilize it with apfloat, you may have to use a Win32 version of apfloat. They should be able to use 2 GB of memory or even more.

If you plan use a non-Intel x86-based platform you should prefer a computer that supports 64-bit integer arithmetic in the hardware (especially multiplication). Suitable processors are for example the DEC Alpha, MIPS R4000 (or greater) and the UltraSPARC. In the 32-bit gcc implementation the basic modint class multiplication uses long long ints for multiplication and remainder. This will be slow if emulated in software with a poor compiler. The 64-bit implementation doesn't use integer division hardly at all and is really fast for example on the Alpha. The floating-point versions should be preferred only on computers with extremely good floating-point performance and abysmally poor integer performance, or when a precision of more than 60 billion digits is required.

The general 32-bit version doesn't use 64-bit integer arithmetic but both 32-bit integer and floating-point arithmetic. It might be faster than the 32-bit gcc version if your processor converts fast between floating-point and fixed-point data types, so you might want to experiment.

7.1 Assembler Optimization

Hand-optimizing the code in assembler makes only sense when the compiler can't optimize the code well. On RISC processors the compiler generates mostly better code than what could be achieved with hand-optimization, since it automatically checks instruction scheduling and other complicated things. Using assembler thus is useful only when special hardware-specific instructions need to be used (like single to double-width integer multiplication).

However on older processors, which were not designed for easy and effective C compilation, like the Intel x86 series, optimizing the critical parts in assembler can make the program several times faster. This is obviously due to the small number of registers, bad instruction set and overall ancient and inefficient processor design.

7.1.1 Modular Multiplication

The modint class multiplication (multiplication and then remainder) in the Number Theoretic Transforms (see appendix A) is the most time-consuming single operation of the whole program. That is why special attention was paid to making it as fast as possible.

Some processors, like the Intel x86 family, have special double-width multiplication and division instructions in the hardware. That is, you can multiply two 32-bit integers and get the whole 64-bit result, and divide a 64-bit number by a 32-bit number, supposing the quotient and remainder fit in 32 bits. This seems naturally to be a very good scheme for doing the modular multiplication, and is actually the best for 386 and 486 processors.

The modulo reduction can be done very effectively in some cases when the modulus is of suitable form. For example, if the modulus is $p=2^{64}-2^{32}+1$ (which is prime and of suitable form for radix-2 Number Theoretic Transforms up to length 2^{32}) the remainder of a 128-bit result (from multiplying two 64-bit numbers) can be done with a few shifts, additions and subtractions. Now suppose the result of a multiplication is $2^{64}A+B$ (for example if the result is stored in two 64-bit registers, A is the contents of the upper word and B is the contents of the lower word). Then

$$2^{64}A+B \equiv (2^{64}-2^{32}+1)A+(2^{32}-1)A+B \equiv (2^{32}-1)A+B(\mathrm{mod}\ p)$$

which is a shift (or two), an addition and a subtraction. Then the result is about 96 bits and the operation can be performed again, which should produce a result of about 64 bits. Checks for overflows might require a few extra instructions. This should in general be faster than a normal division instruction (if available), and extremely fast if implemented in special hardware.

This scheme is implemented in the 64-bit version of the kernel of this program, since there exist three suitable primes for the Chinese Remainder Theorem: $2^{64}-2^{40}+1$, $2^{64}-2^{34}+1$ and $2^{64}-2^{32}+1$. With the two first primes the shifting scheme needs to be done three times to reduce the remainder to 64 bits. This scheme doesn't work very well for 32-bit numbers. First there exist only two primes of the suitable form: $2^{32}-2^{30}+1$ and $2^{32}-2^{20}+1$. If more powers of two are added to or subtracted from the modulus, the number of instructions grows and a general-purpose division will be faster. If the middle power of two (like 2^{30} in $2^{32}-2^{30}+1$) is very close to the word size (2^{32}), very many shifts are required and again the scheme becomes slow. Second, for reasonable transform lengths the middle power of two should be relatively large (since must be $p=kN+1$ where N is the transform length) which makes the scheme useless, since there simply doesn't exist suitable primes.

Another drawback is that the transform length must be a power of two which is not the case for the Winograd Fourier Transform Algorithms (see [9], [10]). The WFTA is actually not used in this program but the package includes optimized routines for the transform.

A general division approach which enables an arbitrary modulus was used in the 32-bit core version of the program. Unlike the 386 or 486, the Pentium has a relatively fast FPU in which the modulo reduction can be done more effectively than in the integer unit. Since the division is always done by a constant, it can be replaced with multiplying by the inverse of the divisor. Also the FPU registers have internally 64-bit precision, which makes this scheme possible.

Now if we want the remainder of *a* times *b* divided by *m* the procedure is as follows:

1. Multiply $a \cdot b$.
2. Store the result in a temporary register.
3. Multiply by $1/m$ (precalculated).
4. Take the integer part.
5. Multiply by *m*.
6. Subtract from the temporary result stored in step 2.

The trickiest part is step 4. The x87 series coprocessors have a "round to integer" instruction, which is very slow. When the result is known to be in a suitable range (that is below 2^{63}, which is now the case, since *a* and *b* are less than 2^{31}) the result can be calculated by first adding 2^{63} to the result and then subtracting 2^{63}. This simply makes the fractional bits to be shifted out, since the mantissa's width is 64 bits. The processor's rounding mode must be first set to truncation so that the fractional bits are simply discarded.

A question that arises now is that can round-off errors cause the result to be incorrect. For example, when calculating with a finite wordlength calculator first $1/3 \approx 0.333$ and then multiplying $3 \cdot 0.333 \approx 0.999$ and taking the integer part, one doesn't get 1 as expected but 0 instead. It's easily seen that this cannot happen in the method used. First, *m* is always prime. Second, *a* and *b* are less than *m* (and nonnegative). For the result $a \cdot b$, when multiplied by $1/m$, to produce a decimal expansion like 0.999... the product $a \cdot b$ should be divisible by *m*. This is obviously not possible, since *m* is prime. So there will always be a random enough fractional part for the method to work.

Using the FPU the modular multiplication takes about 33 clock cycles on the Pentium whereas using the integer unit it would take about 52 clock cycles.

Also the Pentium's FPU can start one floating-point instruction each clock cycle, but most instructions have a latency of three clock cycles. Since all the steps 1. – 6. above are dependent on each other, it's possible to perform three independent modular multiplications in parallel using about as much time as one modular multiplication would take.

Also the Pentium can execute floating-point code and integer code in parallel. In some parts of the code where modular multiplications can't be overlapped it's possible to perform for example modular addition in the integer unit and modular multiplication in the FPU at the same time. This makes some parts of the code almost twice as fast.

The Pentium Pro/II/III/Celeron series processors perform very well with the Pentium specific optimizations implemented in apfloat. Although the P6 series processors use speculative execution, the execution units themselves are architecturally very similar to those in the original Pentium. The raw integer and floating-point processing power is quite similar to the Pentium, overall performance per clock cycle being slightly higher because integer, floating-point and memory instructions can all be executed at the same time (the original Pentium can execute a total maximum of two instructions per clock cycle).

It is a good question if some P6 specific features could be used to develop a more efficient FNT algorithm. The P6 has a more efficient and fully pipelined integer multiplication unit, for

example. The Pentium specific version already avoids unpredictable conditional branches, and using the new conditional move instructions appear to bring no noticeable performance improvement over the current code. Also changing the nested loops in the FNT to a single loop, to avoid mispredicted branches, seems to have no significant effect. Currently, the Pentium specific versions of apfloat should always be used with Pentium Pro/II/III/Celeron processors.

7.1.2 Modular Addition and Subtraction

Addition and subtraction are also extensively used in the Number Theoretic Transforms. Since the calculations are done modulo the modulus, an intuitively appealing scheme for addition is to add the operands, then compare the result to the modulus and if the result is not less than the modulus, subtract the modulus from the result. This would seem to require a conditional jump: if the result is less than the modulus, jump past the next instruction, which would subtract the modulus from the result.

Most modern processors have some kind of a branch prediction system, which predicts whether the conditional branch will be taken or not and the following instructions are fetched from the predicted address into the pipeline. This logic is usually based on how the branch behaved before. In a loop for example, the branch is always taken and thus it is not very difficult to guess that the branch will also be taken the next time. Processor manufacturers often report that the branch prediction logic is correct more than 95% of time. This might very well be true, since most code on average is loops or other parts of code which is executed the same way over and over again. However, in the Number Theoretic Transform the conditional branch is totally random, since the numbers tend to be totally random and 50% of time the branch is taken and 50% of time it's not. Thus the branch prediction logic will be wrong about 50% of time. Most processors that have a branch prediction unit behave very badly when the branch prediction logic is wrong. It can take ten clock cycles to clear the whole pipeline and fetch new instructions from the correct address. Thus it would make sense to avoid a random conditional jump at all costs.

Processors that were designed to be superpipelined (like the DEC Alpha series) have conditional move instructions that eliminate this kind of situations where the pipeline is bound to stall. Since the compiler optimizes the code to use these instructions, the problem mentioned above is obviously avoided. However older processor designs, like the Intel x86 series (except the Pentium Pro and Pentium II/III), don't have this kind of instructions. The modulo calculation can be done without branching, but it takes a few extra instructions. The procedure for modular addition on the x86 processors is:

1. Add the operands.
2. Compare the result with the modulus. If the result is greater than or equal to the modulus, the carry flag is set.
3. Subtract the carry flag from zero. Now the result is zero if the result of the addition was less than the modulus, otherwise the binary representation is all ones.
4. Logical and the result from step 3 with the modulus. The result is the modulus if the result of the addition was not less than the modulus, otherwise zero.
5. Subtract the result of step 4 from the result of the addition.

For modular subtraction the procedure is similar, but the carry flag is automatically set if the result of the subtraction is less than zero and the modulus (or zero) is added to the result of the subtraction.

The operations above are highly dependent on each other. Since the Pentium executes two independent instructions on each clock cycle, the code can be made about twice as fast when modular addition and subtraction are calculated in parallel. This is always the case in the Number Theoretic Transforms.

7.1.3 A Note for Pentium and "Pentium" Users

Please note that the assembler optimization for the Pentium processor is really done exclusively for the genuine Pentium. There are many Pentium (or 586) clone processors out there (NexGen, Cyrix, AMD etc.) which may behave totally differently from the Pentium. Apfloat's assembler optimization uses the Pentium's instruction pairing ability as effectively as possible. The code might be far less efficient on another processor. Also most of the clone 586s have a slower floating-point unit than the Pentium (and possibly a faster integer multiplication unit), so it's highly recommended to also test the 486 version of the program and see which one runs faster if you have a non-Intel 586/686/whatever processor.

7.2 Hardware without a Double-Width Multiplier

Doing modular multiplication and especially the Chinese Remainder Theorem effectively essentially requires hardware capable of doing double-width integer multiplication (e.g. a multiplier that produces the full 64-bit product of two 32-bit integers). Since standard C gives no simple tools to do this, the obvious solution would be to use assembler. When a general implementation in C is required, there is a workaround, however.

The standard multiplication operator gives trivially the lower word of the result of the multiplication. Acquiring the upper word is a bit trickier. Now, the floating-point unit of the computer always calculates with the most significant bits of the numbers used. So the upper word of the product can be calculated easily by converting the operandi to doubles, multiplying them, subtracting the lower word of the product (from the integer multiplication), multiplying by 2^{-32} and converting to an integer. Note that it's necessary to subtract the lower word of the product from the product, since we don't know how the computer will round the values used. If it truncates, it's not necessary. Most computers round to the nearest value, however, so if almost all of the lowest bits of the product are ones, the resulting upper word might be too big (since the fractional one bits don't get truncated, but rounded upwards). An even better solution is to only subtract the most significant bit of the lower word of the product. The round-off problem can't happen if it's zero, so this way we avoid subtracting too much from the product. Some computers seem to round numbers very unpredictably sometimes.

Since most modern computers have relatively fast floating-point units, this scheme can be quite fast. The integer and floating-point code can even be executed in parallel in some cases. The only bottleneck is converting integers to doubles and vice versa.

When suitable moduli are chosen, double-width multiplication can be quite well avoided in modular multiplication. Since we're only interested in the remainder (which fits in, say, 32 bits), it would make sense to only use the least significant 32 bits in the whole operation. We can get the lower 32 bits of the product from simple integer multiplication. Then we can approximate the quotient of the product divided by the modulus by converting the operandi to doubles and multiplying by the inverse of the modulus (converted to a double). The inverse of the modulus should be slightly rounded down. Now subtract the modulus (integer) times the approximated quotient (converted to an integer) from the product, using only the lowest 32 bits. Since the quotient was approximated and possibly one too small, the result might be about twice the modulus. So when the moduli are chosen to be less than 2^{31}, we can still get the remainder, since the result now fits in 32 bits accurately. Simply subtract the modulus once if necessary.

7.3 Vector and Parallel Computers

This program is not really designed to be used in vector or parallel (super)computers. It's designed for RISC processors and hierarchical cache-based memory systems, typically personal computers and workstations. Performance on vector or parallel computers will probably be close to scalar computers.

A multithreaded version of the NTT is available for some platforms. This constitutes only a part of the program execution time, so it will not enable perfect scalability on SMP (symmetric multiprocessing) systems. However, it should be possible (but not very easy) to implement the CRT, addition/subtraction etc. on a vector or parallel computer. If disk-based numbers are used, the disk transfer speed will probably remain the bottleneck, even if you have a very high performance disk system, such as a RAID farm. Feel free to experiment.

Appendix A: Number Theoretic Transforms in Convolution

This text assumes the reader has some basic knowledge about elementary number theory, like modulo arithmetic. For a good textbook on the subject read [1].

The Discrete Fourier Transform (DFT) of the N-long sequence $x(n)$ is defined as

$$X(k) = \sum_{n=0}^{N-1} x(n) W^{kn} \tag{1}$$

For an ordinary Fourier transform, W is defined as

$$W = e^{-\frac{2\pi i}{N}} \tag{2}$$

where i is the imaginary unit. However, at present, we will not assume anything about W. The inverse transform is

$$x(n) = \frac{1}{N} \sum_{k=0}^{N-1} X(k) W^{-kn} \tag{3}$$

At present we are not interested in the transform itself, but a convolution. The cyclic convolution $c(n)$ of two sequences $a(n)$ and $b(n)$ of length N is defined as

$$c(n) = a(n) * b(n) = \sum_{k=0}^{N-1} a(k)b(n-k) \tag{4}$$

assuming that $a(n)$ and $b(n)$ are treated as cyclic sequences, that is $b(-1)=b(N-1)$ etc. Now the convolution can be computed more efficiently in the Fourier domain. The convolution corresponds to linear (element by element) multiplication in the Fourier domain. That is, to calculate the convolution, first take the Fourier transforms of the sequences to be convolved, multiply the corresponding elements in the transforms and then take the inverse transform.

The Discrete Fourier transform (1) and its inverse (3) can be calculated using the Fast Fourier Transform in about $N\log N$ operations instead of the N^2 operations that the direct calculation would require. If $A(k)$ and $B(k)$ are the Fourier transforms of the sequences $a(n)$ and $b(n)$, the Fourier transform $C(k)$ of the convolution sequence $c(n)$ is

$$C(k) = A(k)B(k) \tag{5}$$

and $c(n)$ can then be calculated with the inverse transform. Now to see what requirements the number W must meet in general for the cyclic convolution to work, substitute (1) to (5):

$$C(k) = A(k)B(k) = \sum_{i=0}^{N-1} a(i) W^{ik} \sum_{j=0}^{N-1} b(j) W^{jk} \tag{6}$$

then use (3):

$$c(n) = \frac{1}{N} \sum_{k=0}^{N-1} C(k) W^{-kn}$$

$$= \frac{1}{N} \sum_{k=0}^{N-1} W^{-kn} \sum_{i=0}^{N-1} A(i) W^{ik} \sum_{j=0}^{N-1} B(j) W^{jk} \qquad (7)$$

$$= \sum_{i=0}^{N-1} A(i) \sum_{j=0}^{N-1} B(j) \frac{1}{N} \sum_{k=0}^{N-1} W^{k(i+j-n)}$$

Now this is obviously equal to equation (4) if and only if

$$\sum_{k=0}^{N-1} W^{k(i+j-n)} = N\delta(i+j-n) \qquad (8)$$

where $\delta(n)$ is the discrete delta function (1 when $n=0$ and 0 otherwise). So the sum in (8) would be N when $j=n-i$ and 0 otherwise. Now let's look at

$$\sum_{k=0}^{N-1} W^{jk} \qquad (9)$$

This is obviously N when $j=0$. Otherwise multiply it by $(1-W^j)$, the result should be zero:

$$(1-W^j)\sum_{k=0}^{N-1} W^{jk} = W^0 + W^j + W^{2j} + \ldots + W^{j(N-1)}$$

$$-W^j - W^{2j} - \ldots - W^{j(N-1)} - W^{jN} \qquad (10)$$

$$= 1 - W^{jN} = 0$$

So $W^{jN}=1$. Since j was arbitrary (in fact $j \not\equiv 0 \pmod N$), obviously W must be an Nth root of unity (also W is not 1 in general). In the "normal" Fourier transform this is of course true as in equation (2). If W is in general an integer or some rational or real number, this criterion clearly cannot be satisfied. However, a suitable W can be found in the field of integers modulo p when p is a prime of the form $p=kN+1$ where k is an integer and N is the transform length. In this case the Fourier Transform is called a Number Theoretic Transform (NTT). For a more thorough discussion about why p must be $kN+1$ refer to appendix B.

So Number Theoretic Transforms are just ordinary Discrete Fourier Transforms but they are done in a different number field. Most of the formulae and algorithms that apply for the DFT also apply for NTTs. The most interesting property is probably that the NTT can be calculated using a "fast" algorithm (Fast Number Theoretic Transform, FNT), like the DFT can be calculated using the Fast Fourier Transform (FFT). For a rigorous development of the FFT algorithm(s), see [7]. Just remember that W is now an integer and all the calculations are done modulo p. For a clue about FFT implementation, see [3].

Number Theoretic Transforms have several advantages over the usual complex Fourier Transforms:

- The transform is obviously real, so when transforming/convolving real data no special tricks are required to avoid using double the space and work needed.

- Since all used numbers are always integers, no round-off errors can occur. This makes possible to transform very long sequences (like $N=2^{46}$) with standard 53-bit resolution. Also all "twiddle factors" in the FNT algorithms can be calculated efficiently using recurrence relations.
- The computation can be done "in parts" and the final result recovered using the Chinese Remainder Theorem. This is useful if the result overflows (the result numbers are only calculated modulo p). Just do the same calculation modulo several different primes of the suitable form and use the Chinese Remainder Theorem on the results. (For an explanation of the Chinese Remainder Theorem and a description on the implementation see appendix C.)

There are also some disadvantages:
- The transform itself has no use (it has no physical meaning like the Fourier Transform represents frequency). So Number Theoretic Transforms are mostly useful for convolution only.
- Long integer arithmetic is slower than floating-point arithmetic on most computers.

If one happens to have a computer with really poor long integer multiplication/division performance but relatively good integer addition/subtraction performance, one might want to use an algorithm with minimum number of multiplications for the calculation of the NTT. These are of course the Winograd Fourier Transform Algorithms (WFTA) ([9] and [10]). The algorithms can be used as is, but since they involve multiplications by cosines and sines (actually always $i \cdot \sin$), some attention must be paid to what they correspond to in the number theoretic field. A hint to this is given in [13].

For example, if we want to calculate $\cos 30°$ and $i \cdot \sin 30°$ we must have a field that has a 12th root of unity (since $30°$ is one twelfth of the full circle). So $p=k \cdot 12+1$. Now let W be a 12th root of unity in the field of integers modulo p. If we would be in the field of complex numbers, W would obviously be $e^{i\pi/6}=\cos 30°+i \cdot \sin 30°$. So we might assume that in the number theoretic field also must hold $W=\cos 30°+i \cdot \sin 30°$. Also it would make sense that always $\cos^2 x+\sin^2 x=1$ or in other words $\cos^2 x-(i \cdot \sin x)^2=1$. Note that there doesn't necessarily exist a fourth root of unity (corresponding to i) in the number theoretic field.

Now if W is the Nth root of unity and

$$W \triangleq \cos \frac{2\pi}{N} + i \sin \frac{2\pi}{N}$$

$$\cos^2 \frac{2\pi}{N} - \left(i \sin \frac{2\pi}{N} \right)^2 \triangleq 1$$

(11)

then (after simple algebraic manipulation)

$$\cos \frac{2\pi}{N} \triangleq \frac{W^2+1}{2W}$$

$$i \sin \frac{2\pi}{N} \triangleq \frac{W^2-1}{2W}$$

(12)

Trying these formulae out with the WFTA algorithms one can see that they really work.

Since most computers today are cache-based systems the FNTs (like any Fourier Transforms) can be calculated more efficiently with the "four-step" algorithm [4]. For a short proof see appendix D. Doing the transform in shorter blocks results in high data locality and thus a better cache hit rate.

Appendix B: Primitive Roots

For a more rigorous and general discussion, see for example [2].

This text concentrates on primitive roots of primes only for reasons of simplicity. From elementary number theory we know that for all nonzero integers a, when p is prime

$$a^{p-1} \equiv 1(\bmod\ p) \tag{13}$$

(From now on we just might suppose that the modulus p is prime). For all prime moduli there exists a primitive root r (actually many). A primitive root r is an integer that

when the integer x goes from 1 to $p-1$,
then r^x (mod p) goes through all the numbers $1...(p-1)$ in *some* order.

The order of an integer a is the smallest positive integer x for which $a^x \equiv 1$ (mod p). So the order of a primitive root (modulo a prime p) is $p-1$.

Since $a^{p-1} \equiv 1$ (mod p) always, it is obvious that if the order of a is less than $p-1$, the order should divide $p-1$. To see this, notice that when you start multiplying $1 \cdot a \cdot a \cdot a \cdot ...$ (mod p) when the result of the multiplication is 1, the sequence starts over again. And when you have done the multiplication $p-1$ times, the result must be 1. So the order of a must divide $p-1$.

To test whether a number a is a primitive root modulo p, we want to know whether the order of a is $p-1$ or less. The first thing to do is to factor $p-1$. This can be done effectively (when $p < 2^{32}$) with a precalculated table of primes less than 2^{16} and simple trial division. Then if

$$a^{\frac{p-1}{f}} \not\equiv 1(\bmod\ p) \tag{14}$$

for all factors f of $p-1$, a is a primitive root modulo p. Note that one only has to do the test for all prime factors of $p-1$. There's no need to check if a to any smaller power is 1, since raising the 1 to some higher power is still 1, so one can just check the highest possible powers.

There are lots of primitive roots for all primes, so finding one by directly testing numbers should not be too difficult. An easy approach is to test prime numbers $a=2, 3, 5, 7, ...$

An example:

Let $p=2^{32}-2^{20}+1$. Then p is of the form $kN+1$, that is needed for doing (Fast) Number Theoretic Transforms up to length $N=2^{20}$. The factorization of $p-1$ is $p-1=2^{20} \cdot 3^2 \cdot 5 \cdot 7 \cdot 13$.

Now start testing numbers $a=2, 3, 5, 7, ...$ and see if

$$a^{\frac{p-1}{2}} \not\equiv 1 (\mathrm{mod}\ p)$$

$$a^{\frac{p-1}{3}} \not\equiv 1 (\mathrm{mod}\ p)$$

$$a^{\frac{p-1}{5}} \not\equiv 1 (\mathrm{mod}\ p) \qquad\qquad (15)$$

$$a^{\frac{p-1}{7}} \not\equiv 1 (\mathrm{mod}\ p)$$

$$a^{\frac{p-1}{13}} \not\equiv 1 (\mathrm{mod}\ p)$$

(the first a for which this occurs is $a=19$).

A root W of order N, that is, $W^N \equiv 1$ (mod p), but $W^n \not\equiv 1$ (mod p) when $0<n<N$, can be calculated with $W \equiv r^k$ (mod p), when $p=kN+1$. So $W^N \equiv r^{kN} \equiv r^{p-1} \equiv 1$ (mod p).

Note that now $W^{N/2} \equiv -1$ (mod p), so the decomposition of the Number Theoretic Transform to a (radix-2) Fast Number Theoretic Transform really works (just like the FFT). To see this, note that $W^N \equiv 1$ (mod p), and so $W^{N/2} \equiv +1$ or -1 (mod p). But $W^{N/2}$ can't be 1, since then W would be a root of order $N/2$, and it isn't.

Appendix C: The Chinese Remainder Theorem

This is basically the Chinese Remainder Theorem algorithm from [6].

The Chinese Remainder Theorem (CRT) gives the answer to the problem:
Find the integer x that satisfies all the n equations simultaneously:

$$
\begin{aligned}
x &\equiv r_1 (\text{mod } p_1) \\
x &\equiv r_2 (\text{mod } p_2) \\
&\vdots \\
x &\equiv r_k (\text{mod } p_k) \\
&\vdots \\
x &\equiv r_n (\text{mod } p_n)
\end{aligned}
\tag{16}
$$

We will assume here (for practical purposes) that the moduli p_k are different primes. Then there exists a unique solution x modulo $p_1 \cdot p_2 \cdots p_n$. The solution can be found with the following algorithm:

Let $P = p_1 \cdot p_2 \cdots p_n$.

Let the numbers $T_1 ... T_n$ be defined so that for each T_k ($k=1, ..., n$)

$$
\frac{P}{p_k} T_k \equiv 1 (\text{mod } p_k)
\tag{17}
$$

that is, T_k is the multiplicative inverse of P/p_k (mod p_k). The inverse of a (mod p) can be found for example by calculating a^{p-2} (mod p). Note that $a \cdot a^{p-2} \equiv a^{p-1} \equiv 1$ (mod p).

Then the solution is

$$
x \equiv \frac{P}{p_1} r_1 T_1 + \frac{P}{p_2} r_2 T_2 + \ldots + \frac{P}{p_n} r_n T_n (\text{mod } P)
\tag{18}
$$

The good thing is that you can calculate the factors $(P/p_k) \cdot T_k$ beforehand, and then to get x for different r_k, you only need to do simple multiplications and additions (supposing that the primes p_k remain the same).

When using the CRT in a Number Theoretic Transform, the algorithm can be implemented very efficiently using only single-precision arithmetic when $r_k < p_k$ for all k. Now calculate first P/p_k and T_k for all k (note that this only needs to be done once). Then calculate

$$
y_k \equiv r_k T_k (\text{mod } p_k)
\tag{19}
$$

for all k. Now the solution is

$$
x \equiv \frac{P}{p_1} y_1 + \frac{P}{p_2} y_2 + \ldots + \frac{P}{p_n} y_n (\text{mod } P)
\tag{20}
$$

Note that multiplying a multiprecision number P/p_k with a single-precision number only requires single-precision arithmetic (supposing your hardware does double-width multiplication). Also the reduction modulo P in the final calculation obviously only needs simple compares and subtractions, since $(P/P_k)y_k$ is always less than P.

Appendix D: The "Four-Step" FFT Algorithm

The Discrete Fourier transform $X(k)$ of the data $x(j)$

$$X(k) = \sum_{j=0}^{N-1} W^{jk} x(j) \tag{21}$$

can be computed more effectively in computers with a cache memory with the "four step" algorithm. Assume that the transform length N can be factored to $N_1 N_2$. Now treat the data like it was stored in a $N_1 \times N_2$ matrix. Assume that the data is stored in the matrix the "C" way, that is the matrix element $\mathbf{A}_{jk}$ (row j, column k) is stored at linear memory address $jN_2 + k$.

The algorithm is as follows:

1. Transform each column, that is do N_2 transforms of length N_1.
2. Multiply each matrix element $\mathbf{A}_{jk}$ by $W^{\pm jk}$, the sign being the sign of the transform and W the Nth root of unity of the number field used.
3. Transpose the matrix.
4. Transform each column, that is do N_1 transforms of length N_2.

Now the proof that the algorithm actually works:

From now on we'll only use linear addresses of the data.

Step 1.

In column k_2 of the matrix a transform of length N_1 is performed. So the appropriate N_1th root of unity for that transform is W^{N_2}. So the element in row k_1 and column k_2 becomes

$$X_1(k_1 N_2 + k_2) = \sum_{j_1=0}^{N_1-1} W^{j_1 k_1 N_2} x(j_1 N_2 + k_2) \tag{22}$$

Step 2.

Multiplication.

$$X_2(k_1 N_2 + k_2) = W^{k_1 k_2} X_1(k_1 N_2 + k_2) \tag{23}$$

Step 3.

Transposition.

$$X_3(k_2 N_1 + k_1) = X_2(k_1 N_2 + k_2) \tag{24}$$

Step 4.

Again in column k_1 of the matrix a transform of length N_2 is performed. So the appropriate N_2th root of unity for that transform is W^{N_1}.

$$X_4(k_2 N_1 + k_1) = \sum_{j_2=0}^{N_2-1} W^{j_2 k_2 N_1} X_3(j_2 N_1 + k_1) \tag{25}$$

Substituting $X_3(j_2 N_1 + k_1)$ from step 3 yields (noting that now $k_2 = j_2$)

$$= \sum_{j_2=0}^{N_2-1} W^{j_2 k_2 N_1} X_2(k_1 N_2 + j_2) \tag{26}$$

Again substituting $X_2(k_1 N_2 + j_1)$ from step 2 yields

$$= \sum_{j_2=0}^{N_2-1} W^{j_2 k_2 N_1 + j_2 k_1} X_1(k_1 N_2 + j_2) \tag{27}$$

Finally substituting $X_1(k_1 N_2 + j_1)$ from step 1 yields

$$= \sum_{j_2=0}^{N_2-1} W^{j_2 k_2 N_1 + j_2 k_1} \sum_{j_1=0}^{N_1-1} W^{j_1 k_1 N_2} x(j_1 N_2 + j_2) \tag{28}$$

which is by changing the order of the summation

$$= \sum_{j_1=0}^{N_1-1} \sum_{j_2=0}^{N_2-1} W^{j_1 k_1 N_2 + j_2 k_2 N_1 + j_2 k_1} x(j_1 N_2 + j_2) \tag{29}$$

Note that

$$(k_2 N_1 + k_1)(j_1 N_2 + j_2) = j_1 k_2 N_1 N_2 + j_1 k_1 N_2 + j_2 k_2 N_1 + j_2 k_1 \tag{30}$$

and since $W^{N_1 N_2} = W^N = 1$ also $W^{j_1 k_2 N_1 N_2} = 1$ and thus the final result can be presented as

$$X_4(k_2 N_1 + k_1) = \sum_{j_1=0}^{N_1-1} \sum_{j_2=0}^{N_2-1} W^{(k_2 N_1 + k_1)(j_1 N_2 + j_2)} x(j_1 N_2 + j_2) \tag{31}$$

Now this is exactly the same as the original Discrete Fourier Transform, when we note that the summation is just the same, only factored to two parts which comes from factoring N to $N_1 N_2$. Simply mark k with $k_2 N_1 + k_1$ and note that j going from 0 to $N-1$ is equivalent to $j_1 N_2 + j_2$ with j_1 going from 0 to $N_1 - 1$ and j_2 going from 0 to $N_2 - 1$ for each j_1 in the inner loop. So $X(k)$ is equivalent to $X_4(k_2 N_1 + k_1)$. ∎

[4] has a more thorough discussion about the subject. Some ideas that were used in this program are for example the "six-step" method, very similar to the "four-step" method:

1. Transpose the matrix.
2. Transform the rows.
3. Multiply by $W^{\pm jk}$.
4. Transpose the matrix.
5. Transform the rows.
6. Transpose the matrix.

This method has the advantage that the short transforms are done in linear memory blocks. This is a requirement for any cache-based memory system to function effectively. The matrix

transposition is a very fast operation and can be performed in place when $N_1=N_2$ or $N_2=2N_1$ and the array fits in memory. When doing convolution only, one can save the last transposition in the forward transform and the first transposition in the inverse transform. [4] describes the algorithm carefully.

A disk-based "two-pass" transform algorithm described in [4] was also implemented. It requires only two passes through the data set and is theoretically very effective. The implementation avoids transposing the whole matrix when doing convolution. This is accomplished by first reading $N_1{\times}b$ blocks into memory so that the blocks just fit in memory. The block is transposed, each row is transformed, the block then transposed again and written back to disk. After the columns the rows are transformed. In the inverse transform the order is reverse.

However, the "two-pass" algorithm is required only when the whole array can't fit in the memory and thus the transform length will be very big. In practice the algorithm is so slow for disk storage that it's useless. (Actually the algorithm is not intended for disk storage at all but only slower hierarchical memory with seek times and transfer rates more typical for memory chips than disk. This is mentioned in [4].)

Appendix E: Algorithms for π

These are probably the most efficient algorithms for calculating π known to man. Proofs and convergence analysis can be found in [8]. All calculations must be done with the desired precision of the final result.

The Borweins' quartic algorithm:

$$y_0 = \sqrt{2} - 1$$
$$a_0 = 6 - 4\sqrt{2}$$
$$y_{k+1} = \frac{(1 - y_k^4)^{-1/4} - 1}{(1 - y_k^4)^{-1/4} + 1} \tag{32}$$
$$a_{k+1} = a_k(1 + y_{k+1})^4 - 2^{2k+3} y_{k+1}(1 + y_{k+1} + y_{k+1}^2)$$

Now a_k approaches $1/\pi$ quartically, that is the number of correct digits approximately quadruples in each iteration.

The Gauss-Legendre algorithm:

$$a_0 = 1$$
$$b_0 = \frac{1}{\sqrt{2}}$$
$$t_0 = \frac{1}{4} \tag{33}$$
$$a_{k+1} = \frac{a_k + b_k}{2}$$
$$b_{k+1} = \sqrt{a_k b_k}$$
$$t_{k+1} = t_k - 2^k (a_k - a_{k+1})^2$$

π is then approximated by $(a_k + b_k)^2 / (4t_k)$. The algorithm has quadratic convergence that is the number of correct digits approximately doubles each iteration.

The Chudnovskys' algorithm:

$$\frac{1}{\pi} = \frac{12}{\sqrt{640320^3}} \sum_{k=0}^{\infty} (-1)^k \frac{(6k)!}{(k!)^3 (3k!)} \frac{13591409 + 545140134k}{(640320^3)^k} \tag{34}$$

The series must be calculated using the binary splitting algorithm to be efficient. A good explanation of the binary splitting algorithm can be found for example in [14].

References

[1]   Kenneth H. Rosen: *Elementary Number Theory and Its Applications*, Third Edition, Addison-Wesley 1993.

[2]   James H. McClellan: *Number Theory in Signal Processing*, Prentice-Hall 1979.

[3]   William H. Press et al: *Numerical Recipes in C*, Second Edition, Cambridge University Press 1992.

[4]   D. H. Bailey, "FFTs in External or Hierarchical Memory", *Journal of Supercomputing*, vol. 4, no. 1 (March 1990), p. 23 – 35. Also available from `http://www.nas.nasa.gov/NAS/RNRreports/dbailey/RNR-89-004/RNR-89-004.o.html`

[5]   David H. Bailey: "The Computation of π to 29,360,000 Decimal Digits Using Borweins' Quartically Convergent Algorithm", *Mathematics of Computation*, Volume 50, Number 181, January 1988, Pages 283 – 296. Also available from `http://www.nas.nasa.gov/NAS/TechReports/RNRreports/dbailey/pi/pi.o.html`

[6]   Henri J. Nussbaumer: *Fast Fourier Transform and Convolution Algorithms*, 2nd edition, Springer-Verlag 1982.

[7]   E. Oran Brigham: *The Fast Fourier Transform*, Prentice-Hall 1974.

[8]   J. M. Borwein & P. B. Borwein: *Pi and the AGM⸺a Study in Analytic Number Theory and Computational Complexity*, Wiley 1987.

[9]   S. Winograd: "On Computing the Discrete Fourier Transform", *Mathematics of Computation*, Volume 32, Number 141, January 1978, Pages 175 – 199.

[10]   Harvey F. Silverman: "An Introduction to Programming the Winograd Fourier Transform Algorithm (WFTA)", *IEEE Transactions on Acoustics, Speech and Signal Processing*, Vol. ASSP-25, No. 2, April 1977, Pages 152 – 165.

[11]   Jörg Arndt: The Hfloat Package. Available from `http://www.jjj.de/`

[12]   David H. Bailey: MPFUN:A Portable High Performance Multiprecision Package. Available from `http://www.nas.nasa.gov/NAS/TechReports/RNRreports/dbailey/RNR-90-022/RNR-90-022.html`

[13]   David H. Bailey: "Winograd's Algorithm Applied to Number Theoretic Transforms", *Electronics Letters*, Vol. 13, September 1977, Pages 548 – 549.

[14]   Jörg Arndt, Christoph Haenel: *Pi: Algorithmen, Computer, Arithmetik*, Springer-Verlag 1998.

Revision History

2.33 September 9th, 2001
- Fixes for gcc 3.0 compatibility.

2.32 April 17th, 2001
- Minor bug fixes.

2.31 October 22nd, 2000
- 64-bit versions for Borland C++ and Microsoft Visual C++.
- Minor bug fixes and performance improvements.

2.30 August 13th, 2000
- Parallel processing π calculation program.
- Multithreading FNT algorithms for Win32 and Posix threads.
- Saving and loading numbers to/from disk using the `swapto()` and `swapfrom()` member functions.
- Environment variables for system settings as an alternative to those in the file apfloat.ini.
- Some bug fixes and small performance improvements.

2.21 July 29th, 2000
- Various bug fixes and small performance improvements.
- 64-bit version for Linux.

2.20 July 7th, 2000
- Stream input operators.
- Modulo power function with a sample RSA encryption application.
- Various bug fixes and some performance improvements.

2.10 April 7th, 2000
- Some performance improvements.
- Versions for Microsoft Visual C++.
- 64-bit version for djgpp.
- Some bug fixes.

2.00 February 27th, 2000
- Improved the multiplication of short numbers dramatically.
- Calculating π is more than two times faster now with the Chudnovsky brothers' binsplit algorithm.
- Some minor bug fixes.

1.51 June 16th, 1999
- Some critical bug fixes.

1.50 October 8th, 1998
- Added integer and rational data types.
- Several bug fixes.
- Hopefully some portability improvement, especially with Borland C++ 5.02 and gcc 2.8.1.
- Some performance improvement, especially in the double version.

1.41 September 15th, 1997 (Not publicly released)
- Added a Win32 version with a Windows GUI.
- Optimized the code for bcc32.
- Minor bug fixes and performance improvement.

1.40 July 5th, 1997
- The transform length can have a factor of three, this can make the program sometimes 25% faster.
- `fmod()` and `modf()` functions.
- More portable.

1.33 October 30th, 1996
- The initialization functions `apinit()` and `apdeinit()` are now called automatically at program start and exit.

1.32 October 10th, 1996
- Fixed problems caused by the C++ complex type which is a template class in the newest ANSI C++ draft.

1.31 August 22, 1996
- Added tapfloat class to store transformed apfloats. This makes multiplying by constant apfloats significantly faster.

1.30 July 31, 1996
- Made apfloat faster. Especially the Pentium version is about 50% faster and the Alpha version seems to be almost 200% faster.
- Using different bases is easier.
- Several minor changes.
- Added a more general 32-bit version which doesn't need gcc's long long ints.
- Realized that apfloat is not ANSI C++ compatible (one reason being that there is no ANSI C++ standard, just a draft).

1.20 July 1, 1996
- Support for arbitrary bases (not just decimal).
- Complex number arithmetic.
- Elementary transcendental functions (**exp()**, **log()**, **sin()**, **cos()**, ...).
- Several major and minor bug fixes.

1.10 May 15, 1996
- Added expandability to various types of processors and different raw data types (the **rawtype** data type). The core functions are always in the files raw.h and bigint.cc, rest of the files are common for all versions. Included core files for:
  - 32-bit: djgpp/Linux 486 and Pentium, general Unix gcc, bcc32
  - 64-bit: general Unix gcc, DEC Alpha
  - pure floating-point (doubles or floats): practically any compiler
- Floor and ceiling functions (**floor(apfloat)** and **ceil(apfloat)**).
- Data size and precision are now of type **size_t** so that 64-bit computers can access more than 4GB of memory.
- Slightly improved the 32-bit Pentium version performance.
- Several minor changes.

1.02 May 11, 1996
Fixed a bug in the character string apfloat constructor.

1.01 May 6, 1996
Fixed a minor bug in the character string apfloat constructor.

1.00 March 20, 1996
Initial release.