m3na
A Modula-3 Numerical Analysis Library

Harry George
hgeorge@eskimo.com

September 11, 2007

Contents

1 Introduction

1.1 Purpose

The `m3na` library is a collection of arithmetic operations on a range of mathematical objects:

1. `Integer` - machine size integers
2. `BigInteger` - integers of arbitrary size
3. `Float` - machine oriented floating point numbers
4. `Complex` - complex numbers
5. `Polar` - 2D vectors in polar representation
6. `Fraction` - fractions of arbitrary objects
7. `Vector` - vectors of arbitrary size
8. `Matrix` -
9. `Polynomial` -
10. `Root` - roots of polynomials
11. `PhysicalValue` - numbers equipped with physical units

It includes algorithms for special functions, fourier transform, interpolation, root finding, random numbers as well as routines for generating LaTeX output and function plots.

All datatypes presented except of the floating point numbers allow for exact computation. That's why one can no longer talk of a library that is specialized for numerical analysis. Though any instantiations of one of the composed datatypes with floating point numbers requires algorithms specialized for the problems of numerical operations.

There are many libraries around that deal with mathematical operations, computer algebra and function plotting, so what is the sense of a new library of this kind? The reason is simply to provide a mathematical tool in the static type safe, logical, comfortable but efficient programming environment of Modula 3. Even expensive computer algebra tools like Maple or Mathematica or wide-spread numeric tools like MatLab can't provide static type safety. Instead they declare the missing static typing as advantage but in practice it proves to be a source of many programming bugs. Same applies for scripting languages like Python and Perl. C++ and Ada may seem to be alternatives because they provide overloading of operators. In fact this technique is syntactic sugar and if heavily used it let the user completely lose control of which functions are actually called. The unlogical syntax of C prohibits Java and again C++.

The only alternative for programming languages suitable for mathematical computations or programming in general is Haskell and similar functional languages. Haskell provides static type checking with easy accessible polymorphism while short programming texts but reducing the user's control over time and memory consumption.

1.2 History

Modula 3 numerical analysis library was started in 1996 and maintained sinced then by Harry George. Many people contributed to it but for many years there was only few or even no further activity on it. In 2003 I (Henning Thielemann) started on this code and restructured the modules to be more generic. E.g. now it is possible to combine each data type with almost each other, say polynomials with complex numbers as coefficients. Module templates are used to achieve this flexibility while being efficient in runtime but unfortunately not in code size.

The restructured `m3na` library preserves all of the previous functions but it is no longer compatible with the original version. I hope that the greater flexibility will excuse this.

1.3 Present

Because of the heavy restructuring of the library this document is currently not conform to the contents of the modules, unfortunately. Also some funny words may have been produced by stupid find/replace of the old module names by the new one.

The library is still evolving and due to the currently small user base no one can foresigh if the API is appropriate in the general case. That's why everything should be handled with care and scepticism. Still the API may change in order to get a better structure!

1.4 Future

For the documentation I suggest the usage of a documentation extraction tool to simplify updates of modules and their documentations. This tool should capable of typesetting formulas with the quality of LaTeX.

Further extensions of the library will contain interfaces to o make use of well tested state-of-the-art numeric linear algebra algorithms as well as to the function plotting package This way we can make use of advantages in this libraries while accessing them through a safe and comfortable interface.

2 Library Architecture

2.1 Directory structure

2.1.1 `doc` - Documentation

The library is documented with an analysis paper for each implementation module. Trivial implementations are not discussed, but significant algorithms and design tradeoffs are explained in considerable detail. The idea is to demonstrate independent development of the *code*, when working from published algorithms which may have associated copyrighted code in their own right.

A collection of stand-alone PDFs, one for each module, can be generated with `make sections`.

2.1.2 `src` - Library sources

The library is divided into simple mathematical types (integer, float, ...) and composed ones (complex numbers, polynomials, ...). Each type has its own directory containing modules which follow the same guidelines. Let's have a look to some files from the `vector` directory to illustrate that:

`Vector.ig`	main type and basic operations that are imported from `VectorRep`, the interface is designed for maximum genericity.
`Vector.mg`	
`VectorRep.ig`	basic operations which work on arrays instead of references and thus are more robust and efficient to use - This module imports routines from either `VectorBasic.ig` or `VectorFast.ig`
`VectorRep.mg`	
`VectorBasic.ig`	routines that invoke the functional interface of the underlying field, the implementation is more generic and less efficient - Avoid calling this module directly, this will simplify changes in the implementation of this datatype.
`VectorBasic.mg`	
`VectorFast.ig`	routines that invoke built-in Modula 3 operations and thus are fast and compact - Avoid calling this module directly, this will simplify changes in the implementation of this datatype.
`VectorFast.mg`	
`VectorFmtLex.ig`	formatting for output
`VectorFmtLex.mg`	
`m3makefile`	quake code for this directory
`vector.tmpl`	quake templates for instantiating vectors of arbitrary types

The `Fast` versions of some modules was originally designed as workarounds for the missing `<*INLINE*>` support of the known Modula 3 compilers. In fact inlining is hard to implement in a compiler so this pragma is accepted but ignored by all current Modula 3 compilers. Since these modules makes use of the infix operators plus, minus, and so on, they are instantiated for floating point types only. Later they were also extended by specific numerical code.

Unfortunately it is not easy to find a module naming convention that does not collide with existing standard modules but guarants for easy usage. E.g. the Modula 3 inventors suggested for each data structure to put the main type along with common operations into one module. E.g. there is a module called `LongReal` which contains type `T` and some basic operations that allow the usage of `LongReal`s as part of `List`s and so on. But we need more basic operations like addition, subtraction, multiplication . . . . Thus I have created a new module with suffix `Basic`. Now the situation is that some modules have this suffix and some have not. This makes the handling of instantiation procedures for `quake` a nightmare.

2.1.3 `test` - Test suite

The test suite is composed of a general driver and a collection of test modules, one for each implementation module. Names of test modules are composed from the string `Test` and the type name, e.g. `TestVector`.

2.2 Module Templates

For `m3na/src` there are `template.i3` and `template.m3` which can be instantiated by copying to and then replacing `XYZ` with the new module name.

For `m3na/test/src` there is `template.m3` which can be similarly copied and edited to make new test modules. Also, that template has a `PROCEDURE TestABC` template.

2.3 Algebraic structures

When considering a set of (mathematical) objects with operations that can be applied to them and the characteristics that the operations offer we talk about an algebraic structure. Most of the data types that are implemented here are algebraically rings. For composed data types it is important to know the capabilities that are required for the element types.

The most common operations are:

`New`	make one
`Copy`	given one, make another one
`Lex`	make one by reading a text string (not implemented yet)
`Fmt`	use one to make a text string
`Compare`	relational operators (can be provided only by ordered types)
`Add`, `Sub`, `Mul`	basic operations that are supported by most types
`DivMod`	sometimes a division is not possible, but the decomposition into a sum of a product and a remainder is possible in some way in most cases

All functions allocate new memory for their results. This allows for easy functional without worrying about temporary results. It requires more allocation and deallocation operations and more temporary memory consumption but I hope that the garbage collector can handle this efficiently. This technique is more safe because you can consider most datatypes as immutable, that is the content of an object is not changed, which removes the problems with functions with side effects.

In many cases the operations have neutral elements:

`Zero`	such that for all x holds `Add(Zero,x)=x`
`One`	such that for all x holds `Mul(One,x)=x`

2.4 Naming

Some rules for chosing identifiers:

1. The main type is called `T` and the module gets the name of the implemented data type. Using qualified identifiers like *Module*`.T` makes clear what `T` actually means. This follows the conventions of `libm3` and thus allows for generic templates.
2. Procedure names start with a capital letter.
3. Procedure arguments are mostly named `x`, `y` for easy Cut&Copy.
4. When writing new procedures, look for similar ones in other modules for adapting argument order and naming.

2.5 Quake

The `quake` utility which is the counterpart to `make` for Modula 3 is a central part of current Modula 3 compiler packages. Without `quake` it would be a nightmare to work with generic modules, since quake generates and incorporates modules that instantiate generic ones for you. Every data type in the `m3na` library is equipped with `quake` procedures stored in *type*`.tmpl` files. that instantiate the data type for new element types. It is important to respect some conventions when writing generic modules and corresponding `quake` procedures.

- Module names contain almost no abbreviations for clarity. Names of instantiated modules can become rather long, so abbreviate them when importing.
- Names of instantiated modules follow a fixed scheme to provide for automatic file name generation. It also allows you to easily remember the name of such a module. The main name is the last part, each element name is prepended. E.g. `BigIntegerFractionPolynomial` operates on polynomials which have coefficients that are fractions of integers of arbitrary size.
- The instantiating helpers from `libm3` expect the specification of all element modules, e.g. `Table(key,elt)`. This was the approach for `m3na`, too, for some time, but it turned out to be to expensive. Instead only the element names are given and additionally arguments specifying the flavor of some modules (like `"Fast"` or `"Basic"`). The `quake` template procedures internally constructs the filenames of all required modules. Be prepared to build additional modules to match all requirements.

  It is possible to give a name that differs from the long naming scheme, e.g. calling a module `BigIntFracPolynomial` instead of the long form. However it is not recommend to break the scheme since it makes further instantiating even worse.
- Abbreviate generic formals and imported modules following the same scheme. Each name component is abbreviated with a beginning capital and maybe following lowercase letters. Neglict implementation extensions like `Fast` and `Basic`. Examples:
  - `LongRealBasic AS R`
  - `LongRealTrans AS RT`
  - `LongRealFmtLex AS RF`

  - `RootBasic AS Rt`
  - `Vector AS V`
  - `ComplexVector AS CV`
  - `ComplexVectorFmtLex AS CVF`

- Generic formals should be ordered in increasing complexity. That is simple types first, other types that have the first type as element follows and so on. Modules that add other functionality to the same type are placed immediately after the basic module. Example: `GENERIC INTERFACE Mighty(R,RT,RF,C,CT,CF,V,VF,CV,M,MF);`

Remark: (where is a good place for it?)
Don't rely on order of RECORD entries, e.g. in TexStyle, FmtStyle, but access them by name.

Avoid VAR parameters for output of results, because the user has to provide dummy variables, if he doesn't need the result. Furthermore VAR paramaters don't make clear that it is only a result. Use RECORD results instead, see `DivMod` for example.

2.6 Arrays

An array is a collection of items, all of the same type, which can be accessed by an index into the array. Array types can be distinguished by their cell type. Further on they could be distinguished by their meaning. For example, a Vector is not really a general purpose array – Instead, it supports certain operations such as a dot product. Similarly, a polynomial might be stored in an array, but it has specific operations such as evaluation, or differentation.

We could distinguish these special uses of arrays by BRANDing them. (I [HGG] tried that for a while.) But it is convenient to allow structural type matching, so that one "Fmt" command works for simple arrays as well as Vectors and rows of Matrices. Therefore all these types are being implemented as arrays. Modula-3 provides several ways to implement the concept of an array. We are generally using dynamic arrays. There is (or may be) some overhead for dynamic linking, and some for allocation, but the payoff in readability is high.

In a few cases (e.g., interpolation tables), we can expect the array to be static or constant. So the appropriate formal is an open array. If a dynamic array happens to be used as the actual parameter, it can be dereferenced prior to the call:

```
y:=LongRealInterpolation.Linear(xarr^,yarr^,x);
```

Modula-3's dynamic arrays only range 0..n-1. But many matrix notations call for 1..n or perhaps some offset into the array. The zero-based indexing has proven to be the most universal. There is no support for other schemes.

3 Concepts

3.1 Overview

Numerical Analysis takes place within the context of a discrete computing machine with finite register sizes. To be specific, we can use the Intel 386/387 as the least common denominator for Modula-3. Further, Modula-3 is currently implemented with a 32-bit Word and IEEE 32-bit and 64-bit floating points numbers, which happen to mirror the 386/387 hardware. A large part of numerical analysis is tuning algorithms so that they are not tripped up by the finite limitations of the hardware.

First we must understand the number representations. Then we can discuss the typical errors which can arise. Next we will discuss ways around them. Finally, we will layout an approach to numerical analysis routines.

3.2 Arithmetic Logic Unit (ALU)

See Craw87, pg 18.

An ALU is the integer heart of a typical CPU. It handles raw machine word operations like:

- rotate
- shift
- bitset and reset
- compare
- test for sign
- test for carry

It also does "mathematical" cardinal and integer operations like:

- add
- sub
- mul
- div

The ALU's 32 bits are typically drawn as 0..31 bits, with the most significant figure on the left.

31	30	29	···	2	1	0

The raw values in the register are typically as given as bitstrings of "0" and "1". Modula-3 conveniently allows this to be shown as a literal, preceded by the radix marker "2_". You don't have to show the 0's to the left of the most significant 1, but it doesn't hurt. Often for formatting, it is a good idea to left-pad with leading 0's to a given width:

```
2_10101010101010101010101010010101 (*all 32 bits used*)
2_10001010 (*just lowest 8 bits used*)
2_00001011 (*just lowest 4 bits are used*)
```

If you do nothing special, these literals will be fitted into a full 32-bit INTEGER when they are assigned to a variable. But by defining types (and variables) with fewer bits, we could save space:

```
RECORD
  i:=2_1010;                    (*32-bit INTEGER*)
  j:BITS  4 FOR [0..15]:=2_1010; (* 4-bit CARDINAL*)
  k:BITS 21 FOR [0..1000000];    (* 21-bit CARDINAL*)
  l:BITS  7 FOR [-5..50];        (* 7-bit INTEGER*)
END;
```

This makes sense in a packed record, but probably not elsewhere. For numerical analysis, Integer.i3 provides integers and cardinals of 8, 16, 32 bits.

Completely independent of the choice of bit lengths is the choice of radix for representing literals. While binary literals help clarify some numerical code, other radixes might help elsewhere. Unlike many languages, Modula-3 allows radixes from 2..16, e.g.,

```
8_03470
16_12AF1
3_012
11_12AB3
```

While raw bits can be used to implement many data types, here we are only concerned with numbers. Unsigned numbers build up as binary numbers, for $0 \ldots (2^{32}-1)$. Signed numbers are represented in 2's complement form, for $-2^{31} \ldots (2^{31}-1)$. See Craw87, pg 7 for details.

The critical thing to know about 2's complement is that the 32nd bit is used to indicate sign. Thus, it is not available for use in building number magnitudes. Modula-3 reflects this by making `CARDINAL` go $0 \ldots (2^{31}-1)$, making it a subtype of `INTEGER`. What about unsigned numbers from $2^{31} \ldots (2^{32}-1)$? You must use Word.T for those, and use the Word module's operations. While these look like function calls (and thus slow), they are really compiled inline as the obvious assembler statisticements.

What can go wrong? Integer matrix is a sure thing, so long as you don't overflow (e.g., go beyond $2^{31}-1$ for `CARDINAL`) or underflow (e.g., go below 0 for `CARDINAL`s or -2^{32} for `INTEGER`s). Modula-3 traps these conditions, but you should plan your algorithm to avoid the conditions. Use multi- precision packages if you need more resolution.

For m3na, we will use `CARDINAL`s when we mean counting numbers (such as for array indices), and `INTEGER`s when we mean quantities which might be negative.

3.3 Floating Point Unit (FPU)

See Craw87, pp 20-30.

The 387 FPU provides IEEE 32-bit, 64-bit, and 80-bit reals. These are structured as a sign bit, an exponent, and a significand (also known as mantissa):

```
32-bit
     sign        =1 bit
     exponent    =8 bits
     significand =23 bits (really 24 due to hidden 1)
     range       =+/- 3.39E38
     precision   =+/-1.18E-38

64-bit
     sign        =1 bit
     exponent    =11 bits
     significand =52 (really 53 due to hidden 1)
     range       =+/- 1.08E308
     precision   =+/- 2.23E-308

80-bit
     sign        =1 bit
     exponent    =15 bits
     significand =64 bits
     range       =+/- 1.19E4932
     precision   =+/- 3.36E-4932
```

There are several problems in using these finite representations for real numbers.

Out of Range Suppose you want to represent one googol (1.0E100). It just can't be done in the 32-bit format. That may seem fanciful for practical problems, but intermediate values (e.g., sums of squares) might go beyond the 32- bit limits.

Representation Error Many decimal numbers do not convert exactly to/from binary. You have to use a representation whose precision is sufficient that this error is not a problem.

Truncation Error Suppose you add two numbers in 32-bit format: $a = 1.0e6$ and $b = 1.00123$.

To get a valid addition, we would need to get both a and b into the significand, both with the same exponent: $(1.0 + 0.00000100123) * 10^6 = 1.00000100123e6$. There aren't enough bits in the significand to represent this. It gets truncated to: $1.000001e6$.

What we need to know is how small one number can be relative to another before part of the smaller gets lopped off. This small amount is usually known as ϵ (epsilon). The value is differnet for various representations, giving EPS32, EPS64, and EPS80. These are calculated as:

```
max = 2^sigbits
rel prec = 1/max = 2^(-sigbits)=~ sigbits * ln(2)/ln(10)

32-bit: about 1.0E-7 = ESP32
64-bit: about 1.0E-17= EPS64
80-bit: about 1.0E-24= EPS80
```

As a consequence, there is no point in trying to tune an iteration solution beyond a certain point, e.g.:

```
IF delta < value*EPS32 THEN
   EXIT; (*can't do any better*)
ELSE
   value:=value+delta; (*room for improvement*)
END;
```

Truncation can happen as in the above example with addition and subtraction. It can also happen with multiplication. This is particularly true for intermediate values (such as squares). It is worth your while to arrange to normalize the calculation toward 1.0 before doing multiplications. Or use a longer real format.

Roundoff Error If you are in danger of truncating, you may round up or down. If the rounding is not done right, it will introduce a bias into the calculation.

Fortunately, Modula-3 reals are specified as IEEE reals, which include extra digits to assure proper rounding.

3.4 32-bit vs 64-bit

Is REAL32 good enough? NR92, pg 25, suggests it is, and in fact uses 32-bit reals for its printed algorithms.

However, Gems90 uses 64-bit reals, despite operating in a speed-driven environment with low end-result resolution (viewable by human eye). Hopkins says:

> "Generally, if the real arithmetic has a mantissa of less than about 30 bits, then a double precision version of the library is produced. In fact the double precision [64-bit] version is by far the most common, since most architectures which mimic IBM floating-point hardware and the IEEE floating-point standard all define the mantissa of real variables to be around 24 bits. This accuracy is not sufficient for reliable numerical computation." Hopk88, pg 34.

Further, the Modula-3 Math library is 64-bit. Thus it appears the choice is REAL64.

3.5 Timing

Each machine will have different timing for its hardware ALU and FPU operations, and each compiler will have different approaches for calling upon these operations (thus further diversifying timing). However, it is probable that for the class of problems we are addressing, we can determine the rough relative timing cost for operations.

Craw87 provides timing data in the appendices. While every combination of source/dest is slightly different, we can use the "best-case" for the register-to-memory operation as the exemplar. If we normalize the cycles to let integer add (32 bit) be "1", and round to integers, the ratios are:

```
iadd32        1
imul32        5
idiv32        6
```

fadd32	3
fadd64	4
fmul32	4
fmul64	5
fdiv32	12
fdiv64	13

In other words, integers are a lot faster than floating- point, and divide is the killer. Interestingly, 64-bit is not much worse than 32-bit. Thus our choice of 64-bit as the basic format is not all that extravagant.

3.6 Development Approach

Here is our approach to developing routines:

1. Pick an interesting topic. If you aren't interested, you won't stick with it long enough to do a good job.
2. Gather relevant sources (e.g., texts, code libraries, journal articles). Determine an effective user interface. Determine test cases from known-good sources such as matrix packages or handbooks.
3. Study the problem until you can explain the standard implementation(s) line by line. You may need to work backwards and forwards from the mathematical formulas to the code. You typically need to work out simple (but not too simple) examples, e.g., 4x4 matrix.
4. Study the iterative operations for truncation conditions. Do you need to renormalize the data toward 1.0? Do EPS checks? Use a longer real format?
5. Implement your version of the algorithm.
   (a) Test it for basic correctness against known good values. Don't worry about being off by a few parts in a thousand. At this stage you should be concerned with wildly wrong results. If it made sense on paper, and you got it to compile, then chances are it is a fencepost error (off-by-one) in the iteration conditions. Debug as needed.
   (b) Test for full accuracy against known-good values. Does your algorithm fall apart with big or little numbers? Is it erratic? Check for truncation errors and more fencepost errors. Debug as needed.
   (c) Test for speed against published algorithms. Use the Time module and multi-second runs.

When it comes time to worry about truncation errors and timing, look for normalization toward the center of the range. The classic is the hypotenuse of a right triangle:

```
hyp:=sqrt(a^2 + b^2); (*where a>b*)
(*a^2 is in danger of producing an overflow
  although the final result
  is perfectly in the representable range.*)

(*Instead, do:*)
r:=(b/a);
hyp:=abs(a)*sqrt(1.0+r*r);
```

When it comes time to worry about speed, consider:

1. Precalculate anything you can:
    - Compile-time constant propagation
    - Runtime initialization
    - Calltime initialization
    - Outer loop instead of inner loop
    - Common subexpressions
2. Unroll loops, especially if you know the algorithm will only need a few iterations.

Do you really need to do these things? Modern compilers know a great deal about such local optimizations. If you can do it and the code becomes more readable and more maintainable, then do so. But don't screw up clean code for a few percent speedup. You will still be debugging when a faster computer shows up on your desk. Instead use your time looking for fundamentally better algorithms. If you still need speed, get out your assembler.

4 NADefinition: Utilities

verbosity, debug

`verbosity` controls the `debug` function. 0 means don't print anything and 3 means print everything. It is up to the user of `debug` to set `level` for each call of `debug`. E.g.:

```
  (*verbosity has been set to 2*)

  debug(1,ftn,"this will print");
  debug(2,ftn,"this will print");
  debug(3,ftn,"this will not print");
```

[HGG: I use `debug` because I don't have m3gdb running.]

Error, Err, and err

The theory is that we carefully define the contract of the interface and expect the caller to live up to it. However, in practice, where it is convenient and not a tremendous overhead we do some defensive editing. We also report exceptions that the caller could not realistically have been expected to know.

We raise just `Error` in this library. It gets an `Err` value, which can be used to do case decomposition in the exception handler.

The selection of `Err` values has been haphazard. We probably should start with the POSIX matrix error enumeration, and grow as needed. Just haven't gotten around to looking at it.

`err` is a shorthand for raising `Error`. Originally, it also printed a message to stderr. Since that is no longer the approach, err is not much used.

There is still an open design decision: How to declare a set of exceptions?

1. One exception for each error - This implies long RAISES lists that are hard to update.
2. One global exception for the library with a number that specifies the detailed error (current approach) - New exceptions can only be added by extending a central module. It's not possible to add detailed error specific information.
3. One global exception with an object parameter. - With subclasses one can achieve a more detailed error analysis. What information should be provided by the exception object? Is handling of exceptions still easy? Is this method efficient?

At least from here on, things are mostly not up-to-date. :-(

A WordEx: Extensions to Word

[Warren D. Smith (wds@research.NJ.NEC.COM) provided the analysis and the module.]

It has often struck me as outrageous that most higher level languages do not allow you to access the fundamental machine-language primitives for add-with-carry, subtract-with-borrow, shift-with-carry, multiply-to-get-a-double-length-product, and double-length-divide-with-remainder. In addition, I think syntactic sugar should be added to m3 for:

SWAP(x,y) Swapping 2 words.

SQUARE(x) Squaring a number.

These are necessary for writing a bignum package or a linear congruential random number generator.

Also, it annoys me that more hardware does not provide the (cray) machine language primitives (very useful for SETs as boolean vectors as words!)

PopCount(x) Returns number of 1s in binary representation of x

FindLeastSignifBit(x) Returns index of the least significant bit of x that is a 1

FindMostSignifBit(x) Returns index of the most significant bit of x (often can be done thru conversion to floating point in normalized form, by the way)

But even if you ARE on such hardware, you can't get at the primitives in your high-level language. Very annoying again. I claim there is a vicious cycle here: hardware designers observe that compiled code has no PopCount() calls so they omit it from their next processor design, and software designers see no reason to provide it since it is not in the hardware! How silly - in fact these are very useful.

So MY SUGGESTION is that these functions be added to modula-3 as an enhancement to the "Word" library, in such a way that the compiler will compile them into the appropriate machine language instructions!!

View this as a selling point of modula-3 - the language that actually helps you to use your machine rather than preventing you!!

However, as a stopgap measure, I provide a software implementation, which the compiler could resort to until such point as the compiler writers provide code-generation support.

[See module for implementations.]

B Integer: Integers

Types

The various short cardinals and integers are for convenience. The Array is used as an index array several places in the library.

fmt, fmtArray

Convenience functions.

Factoring: isprime

This is defined in the xPrime module, but accessed from Integer. [HGG: At one time it was implemented in the Integer module, but compiling the bitmap took too long.]

xPrime is mainly a bitmap of the cardinals from 0..46933. Primes are marked 1 and nonprimes are marked 0. The array is 0-based for processing convenience. 46933 was chosen as the sqrt of the largest CARDINAL ($46933 > \sqrt{2^{31}}$). In factoring problems, we will not get anything larger than this.

The primes themselves were obtained from Project Gutenberg's prime11.txt list. They were converted to the bitmap format via a perl script:

```
#file:     mkprime.pl
#abstract: Make a bitstring of primes
#          for use in Modula-3
#
($sec,$min,$hour,$mday,$mon,$year,$wday,$yday,$isdst) = localtime(time);
print "$0 $mon/$mday/$year  $hour:$min\n";
print "Copyright (c) 1996, Harry George\n";

#---configuration---
$inname ="prime11.txt";
$outname="prime.m3";
$linewidth=60;
$maxline=4850;

#---open files---
open(INFILE,"<$inname") || die "cannot open $inname\n";
open(OUTFILE,">$outname") || die "cannot open $outname\n";


#---skip to line 300---
for ($i=0; $i<299; $i++) {
  $line=<INFILE>;
};

#---read data into array---
for ($i=1; $i<$maxline; $i++) {
  $line=<INFILE>; chop($line);
  push(@primes,$line);
};
$minprime=$primes[0];
$maxprime=$primes[$#primes];

#---generate the bitarray---
###select(STDOUT);
select(OUTFILE);

print "CONST MAXPRIME = $maxprime;\n";
print "TYPE  PrimeMap = ARRAY [0..MAXPRIME] OF BITS 1 FOR [0..1];\n";
print "CONST Primes   = PrimeMap{0\n";

$width=0; $currprime=shift(@primes);
for ($i=1; $i<$maxprime; $i++) {
    #---check linewidth---
    if ($width>=$linewidth/2) {
      $width=0;
      print "\n";
    };
    $width++;

    #---print map---
    if ($i<$currprime) {
      print ",0";
    }
    else {
      print ",1";
     $currprime=shift(@primes);
    };
};

print "\n};\n";
select(STDOUT);

#---fini
close(OUTFILE);
print "min=$minprime, max = $maxprime\n";
```

Numbers beyond 46933 are checked by trying to factor them. If any factor is found, we return FALSE immediately.

If nothing is found up to the number's sqrt, then we return TRUE.

Factoring: factor

[HGG: This is a crude algorithm. Surely there are better approaches available.] We start with the lowest prime and determine (via mod) whether or not it divides evenly into the number being examined. If it is divisible, we collect as many repitities of that factor as we can, and then go on to the next prime. As we go, the number being examined is being divided by its factors, so it is getting smaller. We continually check to see if it has become smaller than the next prime to be checked. When that is true, we quit. Due to the halting criterion, primes are reported as themselves to the 1st power.

We could probably add a test for exceeding the maximum prime in the prime map. In that case, we are (I think) dealing with a prime in the range $\sqrt{2^{31}} \ldots 2^{31}$.

Factoring: gcd

Greatest Common Demoninator is determined by Euclid's algorithm, which is described in many sources. We will reference Sedg88, pp 8-9. Sedgewick gives an implementation in Pascal, and mentions an improvement using mod. We have implemented the improvement.

The idea is to find the largest factor (prime or not) which divides evenly both the numerator and denominator of a fraction. E.g.:

$$\frac{12}{18} = \frac{2*6}{3*6} \text{ thus } gcd = 6$$

sqrt

This comes from Hein96. His example 2b was translated to Modula-3. Heinrich also provides several assembler implementations.

The idea is find $\sqrt{N}$ as the sum of two numbers: $N = (u+v)^2 = u^2 + 2uv + v^2$. P. Heinrich notes that chosing u and v to fit base 2 operations (e.g., shift) allows a fast implementation. Read his analysis for the full discussion.

The approach is based on recognition that $\lfloor \log_2 N \rfloor$ is found from the highest set bit. And $\lfloor \log_2 \sqrt{N} \rfloor = \lfloor \log_2 N^{1/2} \rfloor = \lfloor 1/2 \log_2 N \rfloor$ is the highest set bit in $\lfloor \sqrt{N} \rfloor$. It also depends on the fact that if a number `v` is a power of 2 such that it has only one bit, `vbit`, set, then we can multiply any number by `v` just be leftshifting `vbit` times.

Given this, let $u = 2^{\lfloor 1/2 \log_2 N \rfloor}$, so that u is the value of the highest set bit. Let v be the value of the next lower bit. Then calc $u^2 + 2uv + v^2$. If this comes out $\leq N$ then (at least) the next lower bit must be set. In that case let $u := (u+v)$ and go again. The payoff is in being able to compute $u^2 + 2uv + v^2$ using addition and shifts. If we take $2uv + v^2 = v(2u+v) = v(u+u+v)$, we need leftshift(u+u+v,vbit). NOTE: Heinrich uses `l2` for vbit, but that is easily confused with the number twelve, and the significance may be misunderstood. In our implementation, we use vbit and we reorder calculations to emphasize the importance of vbit and v.

How do we find vbit in the first place? We could start at 31 and leftshift until $u < 0$, indicating that the 31st bit was set. Or we could start at 0 and rightshift until $u = 0$, indicating there were no more set bits. Which one works best depends on where vbit resides on average. We often use integer sqrt's in two places:

1. Graphics, where one might compute a distance as $\sqrt{x^2 + y^2}$. Given x and y in the range 0..1000, the sum is 0..1,000,000, or vbit up to 20.
2. Data analysis, where one might compute $\sqrt{1/N \sum (\bar{x} - x_i^2)}$. Data values here might be from 16-bit ADC's, with an average at mid range or vbit around 15.

This suggests that working from 0 upward is a good approach. See the code comments for details.

Now that we have an approach, what is the interface? Heinrich doesn't statistice a specific domain in his article, but experimentation shows that the algorithm cannot handle inputs whose sqrts are larger than half the bits in the word. In other words, for a 32-bit word, $\sqrt{N} < 2^{16}$. Examining the algorithm, we see this is because u2 is obtained by leftshifting by vbits twice. If vbit is over halfway, we end up shifting data out of the word. Thus, we need to place a range limitation on the inputs.

sin_cos

Coordinate Rotation Digital Computer (CORDIC) began with Volder in 1959. We followed Jarv90, Bert92, and Turkowski's article in Gems90 (pg 494 and code on pg 773). While CORDICs can be used for other transcendentals (see Jarv90), we just do sin and cos. Mainly we follow Bert92, though with bits=16 instead of 14.

The idea is to approximatrixe transcendental functions in ways that require only integer-type operations such as shift and add. The analysis leads to the following equations:

$$\begin{aligned}
\text{bits} &= \text{(whatever you choose)} && (1)\\
\text{base} &= 2^{\text{bits}} && (2)\\
\text{radians\_to\_cordic} &= \frac{\text{base}}{\pi/2} && (3)\\
\text{cordic\_to\_real} &= \frac{1}{\text{base}} && (4)\\
\text{expansion\_factor} &= \prod_{i=0}^{\text{bits}} \sqrt{\frac{2^{2\cdot i}+1}{2^{2\cdot i}}} && (5)\\
\text{init\_val} &= \frac{1}{\text{expansion\_factor}} && (6)\\
j &= 0\ldots\text{bits} && (7)\\
\text{atan\_table} &= \arctan(\frac{1}{2^j})\cdot\text{radians\_to\_cordic} && (8)
\end{aligned}$$

It all starts with the number of bits. We chose 16 as a convenient number. That gives 64K increments per quadrant or 256K around the circle. The numbers were computed in MathCad and copy-and-pasted into Integer.m3. We used Turkowski's idea of hardcoding the arctan table. We used Bertrandom's idea of presetting x to a correction factor prior to starting work.

The algorithm works by trying to coax the angle theta to 0, by adding and subtracting smaller and smaller angles, obtained from the arctan table. In lock step, we adjust x and y by the associated amounts via shifting. In the end, when theta reaches 0, x and y will be the cos and sin in CORDIC units. If we wish, we can divide the x and y values by the base to get real numbers values, though this step is not needed in an all-integer project.

The basic algorithm works in the first quadrant. Jarvis warns that the system doesn't always converge in other quadrants. Bertrandom works only in the first quadrant, and adjusts other values into this quadrant. We followed Bertrandom, but had trouble with his actual quadrant decomposition. As coded in Integer.m3, it works. [HGG: I wonder if there was a typo in the published article.]

C Real: 32-bit IEEE Reals

The basic library architecture is based on 64-bit reals. But it is sometimes handy to use 32-bit reals. And if you are going to use them, it helps to have the standard constants and functions available. We have provided wrappers for the normal Math.i3 functions, in order to hide the type conversions to and from 64-bit reals.

However, that means the 32-bit versions of the functions are *slower* than the 64-bit versions. Eventually, perhaps we should have direct calls to a library of 32-bit real assembler code. For now, Real32.i3/m3 is just a convenience.

D LongReal: 64-bit IEEE Reals

D.1 Types

Why rename LONGREAL? Because numerical analysis is tuned to specific resolutions for specific classes of problems. If you need a 64-bit real to do a job, you'd better be sure that is exactly what you are getting. Of course the language definition says that is the meaning of LONGREAL, but why trust it over time? Maybe Modula-4 will use LONGREAL for the 80-bit reals. By using REAL64, we get to fix it one place. Of course, with structural equivalence, you can still intermingle LONGREAL as desired.

A minor point: Since these routines do not lend themselves to generics, using REAL64 and REAL32 makes it fairly easy to clone a 64-bit routine and generate a 32-bit version. One still has to convert the remaining literals from "D" to "E".

D.2 Constants

These are largely taken from Math.i3 and CRC91. We added others as needed when trying to eliminate literals from routines.

D.3 Math.i3 Functions

Why redirect them to here? Because Math.i3 is inadequate, and Real64 can serve as a replacement.

D.4 Other Functions

sgn

The signum function is occasionally handy. However, for performance purpose, you usually have to do it inline.

D.5 Special Functions

These are implemented in SpecialFunction.m3 but exported via LongReal.i3.

factorial

Following NR92, this is implemented with a cached table for $0! \ldots 34!$. Originally 34 was chosen because it could be handled by REAL32. Having moved to REAL64, we could extend the table, but see no need. Beyond 34! we use the Γ function.

ln_factorial

This one needs a cache also, but it can't be a CONST [HGG: At least, I'm not willing to build it]. So we need a VAR cache, which means it could be in global space, in NT-thread space, or in an object. Since all threads should agree on the table values, I'll go global space. However, this assumes that assigning a real to a table cell is an atomic action. Seems like a safe assumption for me.

How big should the cache be? We typically use ln_factorial when factorial overflows. That goes to 34!, so the cache better be bigger than 34. On the other hand, too big a cache takes extra room (hey, this is Modula-3, so who cares about that? :-), and it takes extra initialization on the startup. We notice that 70! is 1 googol, so that seems a handy cutoff. [HGG: Personally, I've never run into a problem using googols.]

gamma, ln_gamma

The basic idea is to implement Stirling's formula for gamma(n+1). That is described in Grah81, pg 112, and in CRC91, pg 351. CRC91 has the most complete treatment:

$$\Gamma(x+1) = \sqrt{2\pi x}x^x e^{-x}(1 + \frac{1}{12x} + \frac{1}{288x^2} + \ldots) \tag{9}$$

However, for numerical analysis, Lanczos (cited in NR92, pg 213, 6.1.5,) provides a better approximatrixion:

$$\Gamma(z+1) = (z+\gamma+1/2)^{z+1/2}e^{-(z+\gamma+1/2)}\sqrt{2\pi} * (c_0 + \frac{c_1}{z+1} + \frac{c_2}{z+2} + \cdots + \frac{c_N}{z+N} + \epsilon)\ (z>0) \tag{10}$$

With all these exponentials and multiplies, the obvious implementation is to work in logs. As NR92 notes, Γ is typically used in multiply and divide situations, leading to normal (small) numbers. So it makes sense to get ln_gamma, and then get other functions from that.

The exponential needs work. First, there is a repeated factor $x = z + \gamma + 0.5$. Second, since we are returning ln, we really want:

$$\ln(x)(z+1/2)(-x) + \ln(\sqrt{2\pi}) + \ln(\text{series}) \tag{11}$$

This simplifies to:

$$\ln(x) * (z+1/2) + (-x) + 0.918938533204673 + \ln(\text{series}) \tag{12}$$

[HGG: I got the 0.918... number from MathCad.]

NR92 handles the coeffs in a loop. We have unrolled the loop here.

gamma_p, gamma_q

Incomplete Gamma Functions. Per Krey88, pg A78:

$$\begin{aligned}\Gamma(a) &= P(a,x)+Q(a,x) && (13)\\ P(a,x) &= \int_0^x e^{-t}t^{a-1}dt && (14)\\ Q(a,x) &= \int_x^\infty e^{-t}t^{a-1}\,dt && (15)\end{aligned}$$

NR92, pg 216, also gives:

$$\begin{aligned}P(a,x) &= \frac{\gamma(a,x)}{\Gamma(a)} && (16)\\ Q(a,x) &= \frac{\Gamma(a,x)}{\Gamma(a)} && (17)\\ &= \frac{1}{\Gamma(a)}\int_x^\infty e^{-t}t^{a-1}\,dt && (18)\end{aligned}$$

In other words, NR92's Q is Krey88's Q/Γ. We will use NR92's treatment, because it leads to the useful:

$$Q(a,x) = 1 - P(a,x) \qquad (19)$$

This is helpful because depending on the ratio of a and x, Q or P could be the better mechanism for computation. That is, Q is flat when x is small compared to a, and P is flat when x is large compared to a. Either case makes numerical analysis difficult. So we will choose via the criteria given by NR92:

```
if x<(a+1) then
     use P approach
else
     use Q approach
end
```

The next problem is of course to determine the $\gamma(a,x)$ for P and the $\Gamma(a,x)$ for Q. NR92, pg 217, provides a series approximatrixion for $\gamma(a,x)$ and a continued fraction approximatrixion for $\Gamma(a,x)$. These deserve helper functions of their own. While we have used the same names for these functions (gamser for the series and gamcf for the continued fraction), these are original implementations.

gamser

$$\gamma(a,x) = e^{-x}x^a\sum_{n=0}^{\infty}\frac{\Gamma(a)}{\Gamma(a+1+n)}x^n \qquad (20)$$

The initial factor can be captured as follows, using ln_gamma(a) which we pass as lga:

```
    factor= exp(-lga-x+a*log(x))
```

Next, we need to iterate the series to convergence. How converged? We will go until the absolute value of the last term is less than eps=5*EPS.

Notice that each additional term can be produced by multiplying the previous term:

```
  term[i+1]=term[i]*x/(a+1+i)
```

To initialize this, we can do the zeroth term by hand, then start iterating. Note that in the each term, $\Gamma(a)/\Gamma(a+1)$ cancels to $1/a$. In the zeroth term that is it. In later terms, we need $(a+1)(a+1+1)(a+1+2)\ldots$ in the denominator. The $(a+1+n)$ item, called a1n, is initialized to $a+1.0$. For the x's, we start with $x^0 = 1$, and then multiply x's onto the term

```
term:=1.0/a;
sum:=term;
a1n:=a;
FOR i:=1 TO MaxIter DO
     a1n:=a1n+1.0;
     term:=term*x/a1n;
     sum:=sum+term;
     IF ABS(term)<eps THEN
          RETURN factor*sum
     END;
END;
```

gamcf

$$\Gamma(a,x) = e^{-x}x^a \left(\cfrac{1}{x+1-a-\cfrac{1\cdot(1-a)}{x+3-a-\cfrac{2\cdot(2-a)}{x+5-a-\cdots}}} \right) \tag{21}$$

Continued fractions are tedious to set up. We start by copying the betacf code and then working out the new meanings of the variables here:

We need to use n in the calculations, so we need to track the iterator with a real, call it m.

```
m[j]=m[j-1]+1.0
b[j]=m*(m-a);  b[0]=0 so actually start with b[0]=tiny;
b[1]=1
a[j]=a[j-1]+2.0;  a[1]=x+1.0-a
the global TINY will work here
eps=5.0e-7
MaxIter=90
```

The initialization conditions are so odd, I decided to set up a[1], b[1] by hand, and then update them at the end of the loop.

```
j=0:
     b0=0
     f=b0=0 ==> f=TINY
     C=f=TINY
     D=0
j=1:
     bj=x+1-a
     aj=1
     D=bj+aj*D=bj=x+1+a
     C=bj+aj/C=x+1+a+1/TINY
     D=1/D=1/(x+1+a)
     delta=C*D=(x+1+a+1/TINY)/(x+1+a)
     f=f*delta=TINY*(x+1+a+1/TINY)/(x+1+a)
          =(x*TINY+1*TINY+a*TINY+1)/(x+1+a)
          = approx 1/(x+1+a) = D
j>=2:
     m=m+1; m2p1=2*m+1
     bj=x+m2p1-a
     aj=m*(m-a)
```

We can turn m2p1 into an add: m2p1:=m2p1+2.0. We can turn x- a into a precalc: xa:=x-a. Then, with m=0 and m2p1=1 at the start:

```
     m:=m+1.0; m2p1:=m2p1+2.0;
     by:=xa+m2p1;
     aj:=-m*(m-a);
```

Next, I see I actually need that x-a business in the initialization, so I can skip the m2p1 stuff and just do

```
     init: m:=0.0; xa:=x+1-a
     m:=m+1.0; xa:=xa+2.0;
```

Then I see that aj=-m*(m-a)=m*(a-m). I'm not sure if the compiler would have caught that, so I'll do it here. But it is a bit obscure, so folks better read this note before fooling with it.

binomial

The formula is:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!} \tag{22}$$

Even if we weren't doing factorials, we would probably want to do a divide and a multiply as logs. So the formula becomes:

$$\exp(\ln(n!) - \ln(k!) - \ln((n-k)!)) \tag{23}$$

Since we already have ln_factorial, we can use that and just take the exponential at the end.

beta

The standard formula is:

$$B(z,w) = \frac{\Gamma(z)\Gamma(w)}{\Gamma(z+w)} \tag{24}$$

However, we implement using logs, which means we can use ln_gamma.

betai

According to NR92, pp 226-227, the incomplete beta function is defined as:

$$\begin{aligned} I_x(a,b) &= \frac{B_x(a,b)}{B(a,b)} = \frac{1}{B(a,b)} \int_0^x t^{a-1}(1-t)^{b-1}dt && (25) \\ &= 1 - I_{1-x}(b,a) && (26) \end{aligned}$$

NR92 provides both a series and a continued fraction approximatrixion, and statistices the continued fraction approach is good for $x < (a+1)/(a+b+2)$. For cases where this does not hold, NR92 recommends using the I_{1-x} formula. The continued fraction form is:

$$\begin{aligned} I_x(a,b) &= \frac{x^a(1-x)^b}{aB(a,b)} \left[\frac{1}{1 + \frac{d_1}{1 + \frac{d_2}{displaystyle 1 + \cdots}}} \right] && (27) \\ d_{2m+1} &= -\frac{(a+m)(a+b+m)x}{(a+2m)(a+2m+1)} && (28) \\ d_{2m} &= \frac{m(b-m)x}{(a+2m-1)(a+2m)} && (29) \end{aligned}$$

The initial factor needs to be calculated using $B(a,b), x$ or $B(b,a), 1-x$, depending on the convergence condition:

```
     if x < (a+1)/(a+b+2) then
          factor = x^a*(1-x)^b)/(a*beta(a,b))
     else
          factor = (1-x)^b*(1-(1-x))^a/(b*beta(b,a))
     end
```

Notice that $\text{beta}(b, a) = \text{beta}(a, b)$, and that $(1 - (1 - x))^a = x^a$. So the two forms are the same except for dividing by a vs b. So let's make the factor just the common part:

$$\text{factor} = (1 - x)^b * x^a/\text{beta}(a, b) \tag{30}$$

This has lots of exps and muls, so we use log forms:

$$\text{factor} = \exp(a \cdot \ln(x) + b \cdot \ln(1.0 - x) - \ln(\text{beta}(a, b))) \tag{31}$$

But notice that beta is already an exp function. So we can use the raw form and save some calls and calcs:

$$\text{factor} = \exp(a \cdot \ln(x) + b \cdot \ln(1.0 - x) - (\text{ln\_gamma}(a) + \text{ln\_gamma}(b) - \text{ln\_gamma}(a + b)) \tag{32}$$

For the second part of the equation, we follow NR92's lead in using the continued fraction approach. And we use the recommended modified Lentz (NR92, pg 172).

Lentz is a bit tricky. To avoid using b for both the ftn parameter, and the Lentz b[j] item, we change the Lenz a's to aj and the Lentz b's to bj. That makes it much cleaner. Because the iteration variable is lost after a loop completes, we return f directly when found, and raise the error if we actually complete the loop.

```
     f=b0=0, so make f=TINY
     C=f
     D=0
     b[0]=0; b[j>0]= 1
     a[0]=1; a[1]=1; a[2m+1>1]=d1; a[2m>1]=d2
```

We have a tricky initialization for j=0,1,2 and then get rolling with j=3. That suggests doing the j0,j1 cases by hand and then starting the loop with j=2. That would be done with a loop on i=1 to Max, with m=2*i and then m=2*i+1 for the pair.

```
j=0:
     b0=0
     f=b0=0, so f=TINY
     C=f=TINY
     D=0
j=1:
     aj=1
     bj=1
     D=bj+aj*D=1+TINY
     C=bj+aj/C=1+1/TINY
     D=1/D=1/(1+TINY)= approx 1
     delta=C*D=C*1=C=1+1/TINY=very large
     f=f*delta=TINY(1+1/TINY)=TINY+1= approx 1
j=2:
     j=2 is equiv of d1, so the d[2m+1) formula applies,
     using m=0
     aj=-(a+0)*(a+b+m)*x/((a+2*0)*(a+2*0+1))=-
     a*(a+b)*x/(a*(a+1))
          =-(a+b)*x/(a+1)
     bj=1
     D=bj+aj*D=1+aj*1=1+aj
     C=bj+aj/C=1+aj/(very large)=1+aj*TINY=approx 1
     D=1/D=1/(1+aj)
     delta=C*D=1*(1/(1+aj))=1/(1+aj)
     f=f*delta=1/(1+aj) = D
```

These are the actual initialization values. After that we can begin the normal 2m, 2m+1 cycle.

erf,erfc

NR92, pg 220, provides:

$$\operatorname{erf}(x) = P\left(\frac{1}{2}, x^2\right)\ (x \geq 0) \tag{33}$$

$$\operatorname{erf}(-x) = -\operatorname{erf}(x) \tag{34}$$

$$\operatorname{erfc}(x) = Q\left(\frac{1}{2}, x^2\right)\ (x \geq 0) \tag{35}$$

$$\operatorname{erfc}(-x) = 2 - \operatorname{erfc}(x) \tag{36}$$

We implement these as is.

E Complex: Complex Numbers

This module has had an interesting history. It started as a reimplementation of NR92's Appendix C. I [HGG] didn't know about the section 5.4, which explains the algorithms, so I had to reverse engineering the code itself. Once I understood the issues, I reimplemented. I also looked for and used alternative references for several of the functions.

Later, I added the transcendentals from Krey88, ch 12, pp765- 766.

In March of 1996, Warren Smith made contact and submitted a wealth of new code, some dependent on his own Complex module. I folded that into the existing Complex, assuring both HGG and WDS conventions could be used. I removed some of my old code and replaced it with WDS implementations. Some of the old code I left as is. Notice that the results depend on structural equivalence in type checking — doing this in a language which required name equivalence would have been grim.

WDS generally documents his work meticulously in extended comments. For other modules I have translated those to TEX format. However, for this module there is relatively little documentation provided, so none is presented here.

add, sub

Trivial

mul

See also Hopk88, pp35-36

```
c3.re:=c1.re*c2.re-c1.im*c2.im;
c3.im:=c1.im*c2.re+c1.re*c2.im
```

div

This is retained as hgg_div, but WDS's implementation is exported.

The basic formula (CRC91, pg 337) is [x is re and y is im]:

$$\frac{(x_1x_2 + y_1y_2) + i(x_2y_1 - x_1y_2)}{x_2^2 + y_2^2} \tag{37}$$

However, this takes 8 multiplies. Also, if x2 and y2 are different in magnitude, there can be truncation errors. So, if x2 is the biggest, we divide top and bottom by 1/x2:

$$\frac{(x_1 + y_1(y_2/x_2)) + i(y_1 - x_1(y_2/x_2))}{x_2 + y_2 * (y_2/x_2)} \tag{38}$$

Notice how the ratio y_2/x_2 repeats. Also, we use the denominator for both the re and im portions. We now have 6 multiplies, and the opportunity to prevent at least some truncations.

abs

See WDS's Magnitude comments.

The basic formula (CRC91, pg 337) is

$$|z| = \sqrt{(x^2 + y^2)} \tag{39}$$

The simple cases of $x = 0$ or $y = 0$ can be resolved right away.

Otherwise, if x and y are different in magnitude, we can have truncation error. In 64-bits, we have $EPS = 1e - 17$, so x/y (or y/x) would have to be that extreme to cause a problem. I'll take that chance.

If the problem does arise, or if one needs a 32-bit representation (where the trouble arises much sooner), do a normalization. See also NR92:

```
x:=|x|; y:=|y|;
if x > y then r=y/x and:

|z| = (x^2+y^2)^1/2
    = [x^2*(1+y^2/x^2)]^1/2
    = x*(1+(y^2/x^2)^1/2
    = x*(1+(y/x)^2)^1/2
    = x*(1+r*r)^1/2, x > 0
```

Now we have $1 + something\_less\_than\_one$, which might still give a truncation error but has a better chance than just adding the squares. It costs an extra multiply. Of course, if $y > x$, then the analysis is reversed.

```
if y>x then r=x/y and:

  |z| = (x^2+y^2)^1/2
      = [y^2*(x^2/y^2+1)]^1/2
      = y*(x^2/y^2 +1)^1/2
      = y*((x/y)+1)^1/2
      = y*(r^2+1)^1/2, y > 0
```

sqrt

See WDS's Sqrt comments.

The basic formula (Krey88, pg 730, eqn 17) is

$$\sqrt{z} = \pm \left[\sqrt{\frac{1}{2}(|z| + x)} + \text{sign}(y) * i * \sqrt{\frac{1}{2}(|z| - x)} \right] \tag{40}$$

This seems to do ok. NR92, pg177 provides a different set of formulas. When time permits, we should do accuracy and timing comparisons.

exp, ln

From Krey88.

powN, powXY

From Krey88.

cos, sin, tan

For cos, the formula is:

$$\cos(z) = \cos(x)\cosh(y) - i\sin(x)\sinh(y) \tag{41}$$

Where:

$$\cosh(x) = \frac{1}{2}(e^x + e^{-x}) \tag{42}$$

Or:

```
ex:=exp(x); ey:=exp(y);
tmp.re:=+0.5*cos(x)*(ex-1.0/ex)
tmp.im:=-0.5*sin(x)*(ey-1.0/ey)
```

Clearly there is opportunity for truncation in the ex and ey terms. Is it a problem? Taking the ex term to be specific: There is truncation if the exponents of e^x and e^{-x} differ by more than 17. This the exponent of e^x can be up to 1/2 of this or $17/2 = 8$. Thus x can be up to $\ln(1E8) =~ 18$. By using MathCAD, we find we can get to $x =~ 20$ before blowing up.

Alternatively, factor out e^{-x}:

```
tmp.re:=+0.5*cos(x)*(ex*ex+1.0)/ex;
```

This may or may not be more stable, but doesn't fundamental shift x's range. We are still in the $|x| <= 18$ range.

Is that enough? CRC91 provides table values to $x = 10$. Let's accept 18 as the top end, document this, and report an exception if we go over that. This will be true of both the x and y inputs.

```
IF ABS(c.re) > 18.0D0 OR ABS(c.im)> 18.0D0 THEN
  RAISE Error{Err.out_of_range};
END;
```

sin is of course similar. Notice that for sin and cos there are no opportunities for ex and ey reuse. We may as well use the Math cosh, sinh functions (hoping they are more robust than our own analysis here).

tan is *sin/cos*: In this case we can probably reuse ex, ey from the various cosh and sinh calculations. For now we however, we will do it naively.

cosh, sinh, tanh

The formula is:

$$\cosh(z) = \frac{1}{2}(e^z + e^{-z}) = \cos(iz) \tag{43}$$

Where e^z is found by

```
ex:=exp(z.re);
tmp.re:= ex*cos(z.im);
tmp.im:= ex*sin(z.im);
```

Once again, we can do a full analysis, or just take the easy way out. If we ever get into a problem space requiring hyperbolics in an inner loop, we should do timing analysis. For now we will take the easy way out:

```
cosh(z) = cos(i*z) = cos(i*z.re + i*i*z.im) = cos(-z.im + i*z.re)

tmp.re:=-z.im;
tmp.im:=+z.re;
tmp:=C.cos(tmp);
return tmp;
```

sin is similar:

```
sinh(z) = -i*sin(i*z) = -i*(-z.im + i*z.re)

tmp.re:=-z.im;
tmp.im:=+z.re;
tmp:=C.sin(tmp);
(*tmp.re = -i*i*tmp.im = tmp.im*)
(*tmp.im = -i*tmp.re* = -tmp.re*)
t:=tmp.im;
tmp.im:=-tmp.re;
tmp.re:=t;
return tmp;
```

pmul, pdiv

If we happen to be in polar form, then multiplication and division are simple. For pmul:

```
tmp.radius:=p1.radius*p2.radius;
tmp.angle:=p1.angle+p2.angle
```

For pdiv:

```
tmp.radius:=p1.radius/p2.radius;
tmp.angle:=p1.angle-p2.angle
```

In addition, we need to normalize the angles to the $-\pi \ldots +\pi$ range:

```
WHILE tmp.angle < -Pi DO
  tmp.angle:=tmp.angle + TwoPi;
END;
WHILE tmp.angle > Pi DO
  tmp.angle:=tmp.angle - TwoPi;
END;
```

NOTE: `arg` and `toPolar` use `atan2`, which does the normalization itself, so we don't need to handcraft normalization code for them.

F Polynomial: Polynomials

We will use the $[a_0, a_1, a_2 \ldots]$ format for the array representation of polynomials. That allows the array index to be the x exponent. However, it makes the bookkeeping for div and deriv a bit tricky, since they have to work backwards.

eval

NR92 recommends using the nested form:

$$a_0 + a_1x^1 + a_2x^2 + a_3x^3 = a_0 + x(a_1 + x(a_2 + x(a_3))) \tag{44}$$

This can be done in a loop, working down from a[nn]:

```
tmp:=p[nn];
FOR i:=nn-1 TO 1 BY -1 DO
  tmp:=a[i]+x*tmp;
END;
tmp:=a[0]+tmp;
```

add,sub

Trivial.

mul

This is not as simple as add and sub, but it it is still mainly bookkeeping. No special insights needed.

div

The classic form is:

$$\frac{A = a_nx^n + a_{n-1}x^{n-1} + \ldots + a_0}{B = b_mx^m + b_{m-1}x^{m-1} + \ldots + b_0} \tag{45}$$

We set it up for longhand as:

```
          Q=quotient
          ------------------
B         | A
           -B*q[qi]
            --------------
            remainder
```

We repeat a pattern, using $b[m] = bmax$ as the divider, getting q's in decreasing order and taking the remainder of the subtraction as the start point for the next division by bmax. We may as well start by copying A into the remainder and using it directly:

```
q[qi]:=r[ri]/bmax
for all the r's, subtract the appropriate q[qi]*b[bi]
```

The problem is really one of bookkeeping, since we are moving backwards in b,r,and q, and the start points keep changing. Points to remember include:

1. Once the order (i.e., max exponent) of the remainder is less than bmax, we can quit.
2. Q's order is A's order - B's order.

Using $A = u, B = v$:

```
(*u/v = q, remainder r*)
un:=NUMBER(u^); unn:=LAST(u^);
vn:=NUMBER(v^); vnn:=LAST(v^); vmax:=v[vnn];
qn:=un-vn+1; q:=NEW(Poly,qn);
r^:=u^;

FOR ri:=unn TO (unn-qnn) BY -1 DO
    qtmp:=r[ri]/vmax;
    q[ri-vnn]:=qtmp;
    ri2:=ri;
    FOR vi:=vnn TO v0 BY -1 DO
      r[ri2]:=r[ri2]-qtmp*v[vi];
      DEC(ri2);
    END;
  END;
```

That is, we work systematrixically down through the remainders. For each we use vmax to find q[qi], where qi started at un- vn and decreases with the r's (which mirror the u's). Once q is found, we use it (in the form qtmp) to calc the subtraction term, which starts with the biggest v and works down to the smallest. For each v[vi], the applicable r term starts with the currently biggest ri (given by ri2), then decreasing ri2 for every vi decrease.

Testing via MathCAD

$$\frac{1 + 2x + 3x^2 + 4x^3 + 5x^4}{.1 + .2x + .3x^2 + .4x^3} \rightarrow q = .6 + 12.5x, r = .9 + .6x + .3x^2 \tag{46}$$

Multiplying back and adding the remainder I got

$$u = .96 + 1.97x + 2.98x^2 + 3.99x^3 + 5.0x^4 \tag{47}$$

Looks like the algorithm works, though the lower coefs are pretty bad.

deflate

See Swok90, pg 124. Given polynomial a, and wanting $a/(x-c) = b$, where a and b are in the form $a_0 + a_1 * x + a_2 * x^2$, we set up the longhand form:

```
            a[n]n*x^(n-1)
          ------------------------------
x-c     | a[n]*x^n +  a[n-1]*x^(n-1) ... a0
            a[n]*x^n + -c*a[n]*x^(n-1)
            --------------------
                     (a[n-1]+c*a[n])
```

Notice the $c * a[n]$ term for the next iteration. Each step has this addition of c times the previous solution coef. The result is a very regular progression backwards from x^n to x^0. At that point we cannot divide any further, and just have the remainder (if any).

```
m:=LAST(a);
b:=NEW(Poly,m);  (*b is 0..m-1*)
b[m-1]:=a[m];
FOR i:=m-2 TO 0 BY -1 DO
  b[i]:=a[i+1]+c*b[i+1];
END;
rem:=a[0]+c*b[0];
```

However, deflation is often used where it doesn't make sense to generate a brandom new "b" polynomial every time. So we need an in-place approach. Since a[] is being used for both the old input data and the new output data, we have to save the coef for current term:

```
(*ann = LAST(a) *)
b:=a[ann]; asave:=a[ann-1]; a[ann-1]:=b;
FOR j:=ann-2 TO 1 BY -1 DO
  b:=asave+c*b;
  asave:=a[j]; a[j]:=b;
END;
rem:=a[0]+c*a[1];
```

Deflation is typically used where c is a root of a, and therefore $rem = 0$. So the last line may not be relevant.

deriv

Given a polynomial with coefs a[]:

```
p =   a0    +a1*x       +a2*x^2     +a3*x^3     +a4*x^4
p'=   a1    +2*a2*x     +3*a3*x^2   +4*a4x^3    +0
p"=   2*a2  +2*3*a3*x   +3*4*a4*x^2 +0          +0
```

By lining them up this way, the polynomial sense of the vector is maintained. That is, p[2] is the coef of x^2 for p and p'[2] is the coef of x^2 for p'.

Let's start with evaluating p(x). The NR92 recommended way to evaluate polynomials is to nest the multiplies. For p(x) we have:

$$p = a_0 + x * (a_1 + x * (a_2 + x * (a_3 + x * (a_4))));\tag{48}$$

In a loop, this works out as:

```
p:=a[pnn];
FOR i:=ann-1 TO 0 BY -1 DO
  p:=a[i]+x*p;
END;
```

Next, we need to handle derivatives of p. We may as well use NR92's notation of pd[], where the index is the derivative count:

```
p=pd[0]; p'=pd[1]; p'=pd[2]; ...pd[nd];
```

In nested form:

```
p :=a0+     x*(a1+     x*(a2+     x*(a3+     x*(a4))));
p':=a1+     x*(2*a2+   x*(3*a3+   x*(4*a4+   x*(0))))
p":=2*a2+   x*(2*3*a3+ x*(3*4*a4+ x*(0+      x*(0))))
```

Notice the factors multiplying the a terms. For the dth derivative we need:

$$(n)(n-1)(n-2)\cdots(n-d+1)a_n \text{ (d<n)}\tag{49}$$

We could do all the derivatives just like p, keeping track of the offsets and inserting the factors:

```
pd[0]:=a[pnn];
FOR i:=1 TO nd DO pd[i]:=0.0; END;
FOR i:=ann-1 TO 0 BY -1 DO
    pd[0]:=a[i]+x*p[0];
  FOR j:=1 TO nd DO
    factor:=1.0;
    FOR k:=i TO i-j BY -1 DO
      factor:=factor*FLOAT(k,REAL32);END;
    pd[j]:=factor*a[i-j]+x*pd[j];
  END;
END;
```

But notice something in the nested form. Starting at the left, a4 keeps moving to the right as another derivative is taken. In fact, the full (factor,a,x) term is very similar to the previous line's term one column to the left. Can we exploit that to reduce the inner loops in the above algorithm?

We need to catch the one-lower-derivative before it gets updated in the current cycle (i.e., the current column). If we worked backwards from the largest derivative, we would be getting both the right a[] term and the right power of x. Ignoring the factors:

```
  p[0]:=a[ann];
  FOR i:=ann-1 TO 0 BY -1 DO
    FOR j:=nd TO 1 BY -1 DO
      pd[j]:=pd[j-1]+x*pd[j];
    END;
    pd[0]:=a[i]+x*pd[0];
  END;
```

Now, what's happening to the factors?

```
init:
  p"=0; p':=0; p=a4
iter3:
  p"=p'+x*p"=0+x*0=0
  p'=p +x*p'=a4+x*0=a4
  p =a3+x*p =a3+x*a4
iter2:
  p"=p'+x*p"=a4+x*(0)=a4
  p'=p +x*p'=a3+x*a4+x*(a4)=a3+2*a4*x
  p =a2+x*p =a2+x*(a3+x*a4)=a2+a3*x+a4*x^2
iter1:
  p"=p'+x*p"=a3+2*a4*x+x*(a4)=a3+3*a4*x
  p'=p
  +x*p'=a2+a3*x+a4*x^2+x*(a3+2*a4*x)=a2+2*a3*x+3*a4*x^2
  p =a1+x*p =a1+x*(a2+a3*x+a4*x^2)=a1+a2*x+a3*x^2+a4x^3
iter0:
  p"=p'+x*p"=a2+2*a3*x+3*a4*x^2+x*(a3+3*a4*x)=a2+3*a3*x+4
  *a4*x^2
  p'=p +x*p'=a1+a2*x+a3*x^2+a4x^3+x*(a2+2*a3*x+3*a4*x^2)
       =a1+2*a2*x+3*a3*x^2+4*a4*x^3
  p =a0+x*p =a0+x*(a1+a2*x+a3*x^2+a4x^3)
       =a0+a1*x+a2*x^2+a3*x^3+a4*x^4
```

Note that p comes out ok (as expected). Note that p' comes out ok too (definitely not expected). p" (and higher derivatives) are in the right shape, but the factorial terms are incomplete. What is the pattern? Let's set up the program and try it out. Use $a[i] = 1$ for all i, so we can see the factorial terms more easily. Experiments on $x = 1, 2, 3, 4, 5$ (comparing to MathCAD) show a pattern: pd[0] and pd[1] are ok. pd[1] is off by $2 = 2!$ and pd[3] is off by $6 = 3!$. In other words all of them are off by a factor:

```
    pd[j]:=factorial(j)*pd[j]
```

It just happens that 0! = 1 and 1! = 1, so pd[0] and pd[1] look ok. It appears we can clean this up by putting a factorial onto pd after computation. NR92 does this with a home-made factorial. Bu this has a hidden integer→float conversion. We may as well use the factorial function, which does a table lookup (at the cost of a funcall). And let's do it for all the pd's since it is low cost to catch j=0,1, and certainly clarifies the analysis.

G Vector: Vectors

I used naive implementations, but so does Gems90. There is a chance of truncation error in abs and dot. Will need testing to determine if there are problems.

H Matrix: Matrices

The matrix computations are implemented naively. Consider Krey88 to be the source, though any number of texts would serve. Notably, Gems90 takes the naive approach for vectors and matrixrices of 3 and 4 dimensions.

Further testing may show that large matrixes need more careful treatment of truncation.

I Root: Roots of Functions

I.1 Quadratics

quadreal, quadcmpx

These address $ax^2 + bx + c = 0$ for real a,b,c and for complex a,b,c. Internally, quadreal calls quadcmpx if the discriminant is less than zero.

We follow the analysis in NR92 pg 184. However, we add generation of alpha and beta terms via:

$$x_1 = \alpha + \beta \tag{50}$$

$$x_2 = \alpha - \beta \text{ therefore,} \tag{51}$$

$$\alpha = \frac{x_1 + x_2}{2} \tag{52}$$

$$\beta = \frac{x_1 - x_2}{2} \tag{53}$$

[HGG: Initially I couldn't find a way to extract α and β from the NR92 approach. I even implemented quadreal in EXTENDED reals (80-bit), to cover potential truncations. Finally I went back and looked for an extraction. The problem of course was that I was initially looking for α and β as sources for x_1 and x_2, not as results.]

I.2 Nonlinear Functions

In one dimension, root finding means finding a value x such that $f(x) = 0$. Generally you need a clue to get started, e.g., from examining a graph. Examination allows selection of a bracketing pair, x1 and x2, which straddles the x-axis (and thus straddles at least one root). NR92 argues persuasively that you should *always* bracket the root before applying a numerical technique.

That brings up the next question. How can you recognize a bracket pair? f(x1) will have the opposite sign from f(x2). Here are some approaches:

```
Given: y1:=f(x1); y2:=f(x2);

a) IF (y1>0.0 AND y2<0.0) OR (y1<0.0 AND y2>0.0) THEN (*bracketed*)

b) IF y1*y2<0.0 THEN (*bracketed*)

c) IF sgn(y1) = -sgn(y2) THEN (*bracketed*)
```

I haven't timed these out, but on the basis of short-circuit evaulation of relationals, and on the basis of no procedure calls and no multiplies, "a" should be the best. I'll use it in the following routines.

Given an arbitrary pair, we can reach out further and further trying to get a bracket, or we can close the gap narrower and narrower. Both can be useful.

bracket_out

Inspired by NR92's zbrac. The idea is to start with two points, expand by the golden ratio iteratively, and see if there comes a time when the y's are of opposite signs.

NR92 uses 1.6 as the growth factor. Just to be different, I'll use the Golden constant. Also, I'll require $x1 < x2$. NR92's algorithm actually works for $x2 < x1$ also, but it isn't as obvious. I think the slight reduction in generality is more than paid back in readability.

bracket_in

Inspired by NR92's zbrak. In testing, I noticed that when the segments just happen to line up exactly on a root, they miss it. So doing different ranges is a good idea, or doing n's which are not multiples of one another. [HGG: All things considered, I like this routine better than bracket_out, but they can work together.]

root_bisect

Inspired by NR92's rtbis. Also covered by Hopk88, pg 67.

The basic trick is to note that the segment sizes get smaller by 1/2 every time, so we can just do:

```
h:=x2-x1;
...
h:=0.5*h;
```

The exit criterion is:

```
IF h<tol THEN RETURN x; END;
```

Each iteration requires a decision on where to go next, based on y values. But to do this, we need to know which direction in x corresponds to which direction in y. So we need to orient the function. NR92 does $f > 0$ at $x + dx$. Just to be different, I'll do the opposite:

```
y:=func(x1);
IF y>0.0 THEN
  x:=x2; h:=x1-x2;
ELSE
  x:=x1; h:=x2-x1;
END;
(*initialize*)
h:=h*0.5; x:=x+h;
FOR i:=1 TO maxiter DO
  y:=func(x);
  IF y<0 THEN
```

```
        x:=x+h;
      ELSE
        x:=x-h;
      END;
      IF h<xacc THEN RETURN x; END;
      h:=h*0.5;
    END;
```

I decided to precalc y1 and y2 in order to do the bracketing check. That led to slight modifications in the initialization. Next, after running into cases of hitting the root dead on, I decided to add a check for y=0. I also had a nice bug in the $h <$ xacc line. I was vaguely thinking h would always be positive, so I skipped doing an ABS(h) for the comparison with xacc. That of course failed, because if func has a negative slope, h is negative. Putting in ABS did the job.

root_brent

Brent's algorithm is used in the netlib matrix libraries. The quadratic formula is given in NR92, eqn 9.3.1. ...9.3.5. But NR92 only gives code, not the algorithm. To demonstrate derivation from the ideas rather than the raw code, I have used more descriptive naming than NR92, and changed the use of temporaries. I also added quick victory checks at the start.

The bracket pair $a \ldots b$ is always the biggest interval of interest. Sometimes we also have a point c which is known be on a's side but a little closer to b (thus giving a smaller bracket). At worst, c is identical to a. So the best shot for the next interval is $b - c$, called diffnext.

The hard part is deciding what to do about tolerances. The problem is that Brent's algorithm is carefully crafted with truncation errors in mind. You can't just go around pulling temp variables out of loops etc. [HGG: Which I did at one point, then thought better of it.]

After building the code, I tested with $x1 > x2$, $x2 > x1$, $x's < 0$, $x's > 0$, and x's straddling 0. Does just fine as long as there is only one root in the x1 ...x2 range. But I can get $root = 0.0$ or a genuine error if there are > 1 roots. I suspect that is due to confusing the c as it moves back and forth between a and b. Maybe I'll look at this more later.

root_newtraph

The basic newton-raphson root finder is analyzed by Hopk88, pg 83 and an algorithm is given in Krey88, pg 953. NR92 also covers it, pg . The formula is:

$$x_{i+1} := x_i - f(x_i)/f'(x_i) \tag{54}$$

Thus, you must be able to provide $f'(x)$.

The various authors agree that this algorithm can go wrong sometimes. NR92 addresses that by using Brent's idea of dropping back to bisection in a pinch. We will follow their lead. The question is, when should you go to bisection? Given brackets a and b, and r for rootnext:

```
a........r.........b
```

Then $(a-r)*(r-b)$ should be positive no matrixter which direction the axis points. That is, it could be 2 positives or 2 negatives. If we get a negative from the multiply, r is out of bounds.

When we need bisection, we can do: root $:= 0.5*(a+b)$. Again, we don't care whether $a<b$ or $a>b$. However, to have a delta for checking exit criteria, we need:

```
tmp:=root;
root:=0.5*(a+b);
delta:=root-tmp;
```

We start with an arbitrary r set at a, and work from there. To assure that a and b remain bracketing, we need to stick to a convention. Let a be such that $f(a)<0$ and b such that $f(b)>0$. Then when a new root is formed, find its f, and set the proper a or b to root.

NR92 also provides tests for converging too slowly. My algorithm just relies on bisection to safely drag the solution to the root. I have not found conditions which make this an issue — if some arise, we'll deal with it then.

J Interpolation: Interpolation

Given a set of x, f(x) pairs (e.g., in a pair of arrays xarr,yarr), and given an x, use interpolation to find y.

There are general purpose approaches, and also some specialized to general polynomials and others specialized to orthogonal approximatrixions such as Chebyshev approximatrixions. Here we consider only the general cases.

The first step is to find the closest xa value as a start point. Then one can interpolationolate from that value and its yarr pair. We can do linear, quadtraic, nth order polynomial interpolations.

interpolation_linear

Pretty straightforward. I found it giving errors of 1–5% for a 10-value sin table, and 0.01% error for a 100-value sin table.

interpolation_polynomial

NR92 describes it on pg 109. In the basic P form, the result is a polynomial with n terms, of which all but 1 (e.g., k) go to zero. So one then has:

$$P(x) = \frac{(x-x_1)(x-x_2)\cdots(x-x_n)}{((x_k-x_1)(x_k-x_2)\cdots(x_k-x_n))y_k} \tag{55}$$

So we need to collect the numerator and we need to find the right k. Eqn 3.1.3 computes these on the fly, col by col. According to several other books, polynomial interpolation is numerically unstable. But NR92's suggests improvements, based on capturing differences (NR92's eqn 3.1.5):

$$D_{m+1,i} = \frac{(x_{i+m+1}-x)(C_{m,i+1}-D_{m,i})}{x_i - x_{i+m+1}} \tag{56}$$

$$C_{m+1,i} = \frac{(x_i - x)(C_{m,i+1}-D_{m,i})}{x_i - x_{i+m+1}} \tag{57}$$

Note that the recurrence slides the whole equilateral triangle into a right triangle form, with the *first* item in each col placed in the C,D arrays at index 1. Thus as we go along the cols, the number of invalid values grows from the bottom of the arrays.

Once we have the full col worth of C and D values per the recurrence, we select which one to use. NR92 uses a cryptic formula:

```
if 2*ns < (n-m) then
  dy=c[ns+1]
else
  dy=d[ns--]
end
```

Here, ns is the index aligned with the input x. Note that after using d we decrement ns. That is, we move further toward the top of the array. But after using c we stay put. We do momentarily peek further down the array as c[ns+1], but ns stays the same.

The next thing to notice is that n-m is the total number of new values in this col. [From there to n we have old data from previous cols.] ns should be centered in this set of values, which are offset from 1. So ns should be $\approx ((n-m)-1)/2$. NR92 replaces the div with a mul by using 2ns $\approx (n-m)$.

What if ns is too small? That means we are too far toward the top of the array. That will be alright once we go to the next col, because the array will slide up and ns will be correctly centered again. But if we are too far down, we need to decrement ns. Also, if we are dead on, we still need to decrement in order to get centered for the next col.

The remaining problem is to understand 3.1.5. First we note that m means the old column, and $m+1$ means the new column we are currently generating. But the easier way to do this is to think of the righthand side c and d as m-1 and the lefthand side as m. Then as we do the big loop for the columns (m=1,2,...n-1), we just need to use the old c and d, and create the new ones in situ. With this understanding, x[i+m+1] means the x[i+m] as seen for the new column.

Next, note that we have a common factor of $(C_{m,i+1} - Dm, i)/(x_i - x_{i+m+1})$. Before we compute that, we need to check to see if the denominator is zero.

Next, we need to know how many of the cells are valid for a given col. We start with n cells, and lose one every time we calc a new col. NR92 recalcs this as n-m every loop. I have pulled it out as col_n. Mainly I did it to make it more readable, but it just might be faster too.

Once we have the loops running m=1…n-1 and i=1…col_n, we need to access the right cells. I made c and d easier by writing them to 0..n arrays and just using the 1…n part. xa is still in the 0…n-1 form, so I have to replace all `i` with `i-1`. I capture the resulting xa accesses as xi for x[i] in 3.1.5 and xim1 for x[i+m+1] in 3.1.5.

To do offsetting (e.g., to do 4 point interpolation out of a much larger table): The idea is to pass start and len with defaults of 0. If $len \neq 0$ then we know we need to do partial access. From there, we use symbolic start and end values to keep track of indexes. C and D still live in their 1…n world, but we now have to offset the xa[] accesses by the xn1 value (which is usually 0).

The relative error on even a 10-value sin table is impressive (on the order of 1.0e-8). It may be complex, but it sure does its job.

K MatrixDecomposition: Simultaneous Linear Equations

There are many, many routines which are useful for systems of linear equations. We only catpure some of the common forms here.

K.1 Tridiagonal Form

The tridiagonal for can be solved by a forward pass and a backward pass. It is therefore worth our while to find ways to get matrixrices into that form.

householder

This implementation is taken from Naka91, pg 263.

matrix_to_arrays

This is a convenience function, in case you have the full matrix form (e.g., from householder) and need to convert to the reduced form.

tridiag

This one took a while to work out. I started with the obvious conditions of A=1x1, 2x2, 3x3. But by stopping there, I couldn't see the pattern. Only when I handworked the 4x4 case, did the recurrence formula stand out. The clearest case in the 3rd row in a 4x4.

We solve each row to this form, where ut2 is known and d2*u3 is (obviously) unknown. u2=ut2 - d2*u3

Then proceed with row 3: a3*u2 + b3*u3 + c3*u4 = r3

u3 = (r3-a3*u2-c3*u4)/b3 u3 = (r3-a3(ut2-d2*u3)-c3*u4)/b3 u3*(1-a3*d2/b3) = u3*(b3-a3*d3)/b3 = (r3-a3*ut2-c3*u4)/b3 Multiply both sides by b3, then divide by the left factor, which we call den for denominator: den = (b3-a3*d3) u3 = (r3-a3*ut2-c2*u4)/den Note that part of this is solvable, and part is not, so split the pieces: u3 = (r3-a3*ut2)/den - c3*u4/den Resolve as far as possible: u3 = ut3 - d3*u4 We are now in the same form that we started from u2, and are ready to proceed with the next row. This continues until the last row, at which time there is no c(n)*u(n+1) term, and we can solve directly. At that point, we have a vector of ut's, but have to subtract out the d(n)*u(n+1) terms. That means a) we have to save the d terms, and we have to work backwards from u(n) to u(1).

Actually, we can cheat a bit and use the existing u vector as the ut vector, since we never need both values at once. So we load up the u vector going forward, and then subtract out the d*u terms going backwards.

L Random: Random Numbers

L.1 Object Structure

For most topics we have used simple Modules. That is because they did not have statistice memory. However, for the random generators we need to remember data between invocations. The selection of which engine, the seed, the current engine settings, etc. are needed. Therefore, we wrap the generators in objects.

Many different generators can be used. See the Random Number Generator (RNGnn) modules. Each of these is subtyped from Random.RandomGen, which in turn allows the generators to be used to drive various deviate generators.

L.2 Deviate Generators

uniform

This just returns the engine's value, possibly with scaling to fit min..max. It checks to see if the parameters really are different from Min and Max before bothering to do scaling.

exponential

This one is trivial. I don't know what NR92 had in mind with the do-while loop. I just return the log, since I'm already checking for 0.0 in the generators.

gaussian

We have implemented this twice, with Warren Smith's version being provided as the default. See the code for his commentary.

The alternative implementation was based on NR92, pg 289. The basic equation is:

$$p(y)dy = \frac{1}{\sqrt{2\pi}} e^{-y^2/2} dy \tag{58}$$

The trick is to use the Box-Muller method to transformed y's:

$$\begin{aligned} R^2 &= v_1^2 + v_2^2 && (59) \\ y_1 &= \sqrt{-2.0 \ln(x_1)} v_1 / \sqrt{R^2} && (60) \\ y_2 &= \sqrt{-2.0 * ln(x_1)} v_2 / \sqrt{R^2} && (61) \end{aligned}$$

Since R^2 is made from squares, and is supposed to range 0..1, we don't need to actually do the sqrt. We just check to see if R^2 is in the 0..1 range.

Obviously the sqrt and Rsq can be pulled out for a tmp value:

```
tmp:=sqrt(-2.0*ln(x1))/R^2
```

Also, R^2 can be used for x1, so let's make a tmp for it:

```
tmp:=sqrt(-2.0*ln(Rsq))/Rsq
```

Now we get v's until we are inside the unit circle, then use the radius squared (Rsq) to that point as x1. We know x1 will be ¡ 1 because we were already inside the circle. Then use the tmp factor to get the associated y's:

```
y1:=v1*tmp
y2:=v2*tmp
```

The function is only supposed to return one of these at a time, so we save one for use later, and set a boolean to the effect that it is ready to use. Since we already have ”start”, we'll use that. When self.start is true, we need to generate two more y's. When it is false, we use the saved one and set for start again.

gamma

See W. Smith's code for commentary.

dirichlet

See W. Smith's code for commentary.

poisson

Not implemented.

binomial

Not implemented.

M RNG01: Random Number Generators – 01

DECSRC

This is just a wrapper for the DEC SRC-supplied `Random.T.longreal`.

ran0

NR92, pg 278, describes Schrage's approximatrixion to Park and Miller's proposed *Minimal Standard* generator. The basic approach is:

```
a) z = a*(z mod q) - r*(z div q)
b) If z <0 then z = z+m
c) To get back to 0.0 --> 1.0, result = float(z) / float(m)
```

`mod` is tricky, because you have to do at least:

$$z \bmod q = z - (z\text{div}q) * q \tag{62}$$

And maybe deal with negatives. I suspect M3's MOD function is fairly complex compared to a simple idiv. So we ought to take advantage of knowing z is positive, and do our own mod.

Notice that z div q can be pulled out as a temp.:

$$\begin{aligned} tmp &= z\text{div}q \qquad (63)\\ z &= a * (z - \text{tmp} * q) - r * \text{tmp} \qquad (64) \end{aligned}$$

Next, remember that dividing by float(m) is more expensive than multiplying by 1.0/float(m), which can be done once at initialization:

```
Mrecip = 1.0 / float(m)
...
result = float(z) * Mrecip
```

Not too surprisingly, this is what NR92 uses. To claim original code, we can use one of the alternate sets of constants. We'll use the first alternative.

NR92 makes a point of telling us to not include the endpoints (0.0 and 1.0). Thus there is a Min value and a Max value. Min should be such that we don't get a 0.0 when captured in REAL64 format. Min can be a few times bigger than EPS. Max is then that close to 1.0, or (1.0-Min), which can be precalculated.

ran1

From NR92. The trick here is to use the ran0 stuff, but use that result to access a shuffle table. The shuffle table (per NR92, pg 281, fig 7.1.1) requires us to take the result modulo the table size. But as noted above, mod can be done inline. Use that to get the shuffled value, and put the old one in its place.

```
ndx:=z - (z div size)*size
z2:= table[ndx]
table[ndx]:=z
```

Bays-Durham further cooks the data by saving z2 and using it for the ndx computation. That means we have to save it and have to initialize it. NR92 initializes it with the first table value. Just to be different, we'll use the 3rd table value. NR92 also warms up the generator with 8 throwaways. We'll pass on that.

N RNG02: Random Number Generators – 02

[Warren D. Smith provided the analysis and the generators.]

The goal is to combine several random number generators to get at least the randomness of the strongest of them, but to do it in such a way as to get nearly the speed of the fastest of them.

NOTE: I use a lot of global variables here to save statistice, necessary in modula3 since C's "statisticic" feature does not exist. Of course these variables are local to the current module. For that reason, please keep this a seperate module from others, e.g. the derived routines for NormalDeviate, etc. should not be in same module as these underlying routines for uniform deviates.

Some component generators: Following G.Marsaglia: A current view of RNGs, pages 3-10 in Computer Science and Statistics, the interface, Elsevier 1985, a fair number of my generators are of the FIBO(a,b,%) type, where $a > b > 0$ are integers such that $x^a + x^b + 1$ is a primitive trinomial mod 2 and % is a binary operation.

Suitable values of (b,a) are tabulated in D.Knuth: Seminumerical methods, Addison-Wesley 1981, page 28, which is extracted from larger tables in N.Zierler & J.Brillhart: Information and Control 13 (1968) 541-554; 14 (1969) 566-569; 15 (1969) 67-69. Some pairs with "a" prime which go beyond the Knuth table, which you can use if you *really* want randomness, include:

```
  (7,127), (15,127), (30,127), (63,127),    (32,521), (158,521),
  (105,607), (147,607), (273,607),     (216,1279), (418,1279)
```

The binary operation % could be: [- mod 1] for floating point values, [- mod M] or [* mod M] for some integer M, or [this NOT recommended; it is same as - only mod 2] XOR.

Then the meta-procedure Fibo(a,b,%) is as follows (it requires an auxiliary arr : ARRAY [0..a-1], initially filled with random seed values, and initially ia=a and ja=b:

```
MetaRandGen() : Type =
  BEGIN
    DEC(i);
    DEC(j);
    IF i<0 THEN
      i := a-1; (* wraparound *)
    ELSIF j<0 THEN
      j := a-1; (* wraparound *)
    END;
    arr[i] := arr[i] \% arr[j];
    RETURN arr[i];
  END MetaRandGen;
```

Period [Marsaglia & Tsay: Lin. Alg. and its Appl. 67, 147-155, 1985]: If % is $(-\text{mod}2^{\text{wordsize}})$, then if at least one of the arr[] is odd the period will be maximal $= (2^a - 1) * 2^{(\text{wordsize}-1)}$ if the characteristic axa matrix T of the recurrence has full order $J = (2^a - 1)$ in mod 2 arithmetic (this assured by the trinomial condition) and 2*J in mod 4 and 4*J in mod 8 arithmetic. That is, if you square T a times, you get back T in mod 2, but do NOT get back

T in mod 4, arithmetic, and if you square T a+1 times, you do not get back T in mod 8 arithmetic. In fact, this criterion will work for any matrix T, not necessarily the fibo(−) one...

If % is [$*$mod2$^{\text{wordsize}}$], which may be implemented in m3 by Word.Times(), then arr[] must be all odd integers, and by considering discrete logs we see that the period is maximal $= (2^a - 1) * 2^{(\text{wordsize}-3)}$ if the $F(a, b, +\text{mod}2^{\text{wordsize}})$ has maximal period $(2^a - 1) * 2^{(\text{wordsize}-1)}$.

Marsaglia's test results for Fibo(a,b,%) generators with (a,b)=(17,5), (31,13) and (55,24):

XOR With %=XOR the fibo fail seven of the tests in Marsaglia's battery. As Marsaglia says in his conclusion "never use XOR." Despite this, these generators keep resurfacing, for example the "r250" Fibo(250,147,XOR) generator of S. Kirkpatrick and E. Stoll, Journal of Computational Physics, 40, p. 517 (1981) and W. L. Maier, "A Fast Pseudo Random Number Generator", Dr. Dobb's Journal #176.

Subtraction With %=-, among the tests in Marsaglia's DIEHARD battery only the "birthday spacings test" (of frequencies of spacings in sorted sets of deviates) is failed, a symptom both of the linear structure and also of the specific subtractive structure of these generators. Note that the Fibo(55,24,-) generator (which fails this test) is to be found in Knuth's book and also is distributed with the DEC SRC implementation of modula-3. Also note that postprocessing Fibo(55,24,-) sequence with a Bays-Durham shuffler, recommended in Press et al. "Numerical Recipes" to fix suspicious generators, will NOT work to make it now pass this test, since the birthday spacings test does not depend upon the ordering of the deviates only their values.

Multiplication With %= $*$ mod 2^{32}, these FIBO gens passed all the tests in Marsaglia's battery.

Note that the -,+ and XOR gens are "linear" and hence theoretically bad and will always fail "empty slab tests" see below.

Note that shift register, Fibonacci() generators with +,- or XOR, and linear congruential generators x ¡– a*x+b mod M, AND linear combinations of such possibly to different moduli, are all "linear" and thus generate d-tuples of random numbers lying on AT MOST $M^{(1/d)}$ hyperplanes in d-space. Consequently, the nonrandomness of a linear RNG with period M is in principle detectable by the "largest empty slab" test in .4*logM dimensions after only N,

```
N > e^(5/2) * (.4*logM)^2.5 * loglogM,
```

random numbers have been generated. Hence: Please do not rely on a linear generator. If you are going, foolishly, to use a linear congruential generator, though, you want a "good multiplier" a mod M. (Bad multipliers will result in even fewer hyperplanes, for example IBM's infamous RANDU generator generated points lying on only 15 planes in 3-space. Even with fairly good multipliers we still get test failures, e.g. Marsaglia's spectrally good multiplier 69069 mod 2^{32} fails his OPSO test, as does the Berkeley PASCAL 62605 mod 2^{29}). It will suffice, for comparatively good behavior in d-space, that

```
(a^d mod M)/M
```

have only small partial quotients in its continued fraction expansion. This should be tested for $d = 1, 2, .., 8$ at least.

Thus for example, consider the prime modulus $M = 2^{35} - 849$. $(M - 1)/2$ is also prime. I found the generator $g = 145683$ by computer search. The continued fraction expansions

```
CF(g/p)=[235852,1,3,6,1,1,5,1,1,2,1,1,1,1,6,2]
CF(g^2/p)=[1,1,1,1,1,1,1,1,21,2,7,1,8,1,2,1,1,2,5,1,1,1,2,1,2,3,12,2]
```

are rather nice, and the $CF((g^x \bmod p)/p)$ for $x = 3, 4, 5, 6, 7, 8, 9$ are also not bad (no partial quotient larger than 23):

```
 [2,1,1,1,1,1,1,1,1,16,1,3,5,1,2,1,1,3,1,1,1,21,1,8,10,5]
 [8,4,3,1,1,3,4,2,7,1,22,22,9,1,12,1,2,2]
 [1,1,1,20,1,23,1,4,2,2,1,1,1,8,1,1,2,1,2,2,1,22,1,2,1,3]
 [1,1,15,1,1,10,2,2,2,1,4,5,1,2,1,1,5,7,7,2,2,4,2]
 [1,1,5,2,2,1,1,1,7,11,3,4,1,9,1,6,3,1,8,1,3,1,1,1,2]
 [1,13,1,7,8,1,1,2,16,2,2,1,1,6,4,1,3,1,1,3,15,2]
 [2,1,2,1,15,2,1,5,1,8,1,1,2,1,4,3,1,3,1,1,1,1,2,1,3,1,3,5]
```

so we conclude that using g as a multiplier should exhibit comparatively good behavior in dimensions 2-9. This particular (g,M) pair has the virtue that IEEE doubles can represent integers up to and including $2^{53} - 1$exactly, so that the modula-3 statisticement

```
 x := (g*x) MOD M;
```

will evaluate it exactly.

As another example, the Marsaglia multiplier $69069 \bmod 2^{32}$, while spectrally good in dimensions 2-5, is bad in dimension 6, as is revealed by the spectral test directly but also simply by noticing that the CF expansion of $69069^6/2^{32}$ is [1, 75, 1, 2, 2, 1, 20, 10, 3, 10, 2, 2, 1, 12, 9] which contains the large number 75 early on.

You also probably want full period M, which happens if [thm page 16 Knuth]

```
  (1) GCD(b,M)=1;
  (2) a-1 is a multiple of p for every prime p dividing M;
  (3) a-1 is a multiple of 4 if 4 divides M.
```

If M is prime and b=0, you get maximal period M-1 if a primitive mod M. If M¿=16 is power of 2 and b=0, get maximal period M/4 if a=3 or 5 mod 8.

Other Fibo generators

Instead of just using one lag term and binary operation %, you could combine with TWO lag terms via some TERNARY operation, or THREE lag terms via a QUATERNARY operation, etc. I suggest the new generator

```
  QuaternaryFibo(a,b,c,d, x0 - (x2 XOR (x1 - x3) mod M) mod M ).
```

If M is 2^{wordsize} and $x^a + x^b + x^c + x^d + 1$, $a > b > c > d > 0$, is a primitive polynomial mod 2, then this generator's period will be at least $2^a - 1$ simply by considering the LS bit, which will follow a DeBruijn sequence and thus exhibits good randomness in $\leq a$ dimensions. My idea is that by using both XOR and +, we hope to avoid the weak behavior of either operation alone, e.g. with respect to the birthday spacings test. This may also make the generator "nonlinear" (i.e. not outputting a lattice), a point I am unsure about. Ternary generators of this type do not seem to exist since 4-term primitive polynomials $x^a + x^b + x^c + 1$ mod 2 do not exist (you need an odd number of terms, as can easily be shown). For tables of examples of primitive polynomials mod 2, see E.J.Watson: Math. of Comput. 16 (1962) 368-9 but the ones given are the lexically first examples, bad for our purposes.

Hence I constructed my own small table below of examples of primitive polynomials $P = x^a + x^b + x^c + x^d + 1$ mod 2, where a is prime and $N = 2^a - 1$ is a Mersenne prime. For such a, P is primitive if P^N divides $x^N - 1$ mod 2, which you can test by the algorithm

```
  Q := x;
  FOR i:=1 TO a-1 do
    Q := Q*Q*x mod P;
  END;
  IF Q=1 THEN "P is primitive."
Table:
x^19 + x^9 + x^4 + x^3 + 1
x^19 + x^6 + x^2 + x^1 + 1
x^31 + x^22 + x^20 + x^8 + 1
x^31 + x^16 + x^11 + x^6 + 1
x^61 + x^5 + x^2 + x^1 + 1
x^61 + x^57 + x^2 + x^1 + 1
x^61 + x^19 + x^16 + x^9 + 1
x^89 + x^36 + x^2 + x^1 + 1
x^89 + x^57 + x^14 + x^5 + 1
x^89 + x^28 + x^26 + x^9 + 1
x^89 + x^37 + x^5 + x^1 + 1
x^107 + x^53 + x^45 + x^12 + 1
x^107 + x^93 + x^68 + x^2 + 1
x^127 + x^29 + x^17 + x^7 +
x^127 + x^67 + x^65 + x^17 + 1
x^127 + x^60 + x^22 + x^9 + 1
x^521 + x^46 + x^38 + x^37 + 1
x^521 + x^249 + x^92 + x^87 + 1
x^521 + x^353 + x^258 + x^6 + 1
x^607 + x^66 + x^60 + x^30 + 1
x^607 + x^247 + x^83 + x^71 + 1
x^607 + x^483 + x^449 + x^298 + 1
x^607 + x^442 + x^113 + x^23 + 1
```

It is a disgusting fact that in modula3 as well as most high level languages, the machine language multiply instruction that computes a*b to double precision is NOT ACCESSIBLE. Ditto "add with carry". This makes an efficient implementation of high precision multiplication (& modular version) obnoxiously difficult and inefficient. And you really do need multiple precision to get a decent period with a lincong or iterated squaring generator. [Note the SQUARE ROOT of the period must be unreachably large for good randomness, NOT the period, a common error.] Here is a MetaAlgorithm which works if numbers 0..modulus*2-2 are representable...:

```
MetaModularMultiply(x,y, modulus: NumType) : NumType =
VAR
  q : NumType := x;
BEGIN
  FOR b = bits of y in MS-->LS order starting
      at the bit AFTER the first 1 bit DO
    q := q+q;
    IF q>=modulus THEN q := q-modulus; END;
    IF b=1 THEN
      q := q+x;
      IF q>=modulus THEN q := q-modulus; END;
    END;
  END;
  RETURN q;
END MetaModularMultiply;
```

One way to improve the randomness of a generator is C.Bays and S.D. Durham's shuffling algorithm, ACMTOMS 2 (1976) 59-64, also described in D.Knuth: Seminumerical algorithms. Knuth says this will output a "considerably more random" sequence. However, the set of values will still be the same (although ordered differently) hence, e.g., the subtractive and XOR fibo gens will STILL fail Marsaglia's birthday spacings test even after improvement by shuffling. However, linear congruential generators are probably substantially improved by shuffling since the lattice structure is destroyed.

L.Blum, M.Blum, M. Shub: A simple unpredictable psuedo-random number generator, SIAM J. Comput 15,2 (1986) 364- following A.Yao: Theory and applications of trapdoor fns, 23rd SFOCS (1982) 80-91 and W.Alexi, B.Chor, O.Goldreich, C.P.Schnorr: 25th SFOCS (1984) 449-457. point out that iterated squaring generator

```
x <--- x*x mod M
```

where M=p*q, p, q primes that are 3 mod 4, will have these properties (assuming it is computationally infeasible to compute the factorization of M)

1. infeasible to predict the PREVIOUS x.
2. infeasible to predict the least signif bit of the previous x with correctness prob $> 1/2 + 1/$polynomial.
3. infeasible to predict the boolean "x¡(M-1)/2" for the previous x with correctness prob $> 1/2 + 1/$polynomial.

Consequence of (2) is: no polynomial time statistical test can invalidate the randomness of the sequence of LS bits of the iterated squaring generator.

The full iterated squaring gen, not just its LS bits, is a lot faster and (conjecturally!) just as random. Its period is (p-1)*(q-1)/4.

Example: Here is a product of two primes both 3 mod 4:

```
94906247 * 94906219 = 9007193062250093 = 2^53 - 6192490899.
```

Here is a smaller example:

```
    M = 9739 * 9719 = 94653341.
```

This modulus has the virtue that iterated squaring mod M may be accomplished exactly in IEEE double arithmetic using

```
 x := x*x MOD M, since M^4 < 2^53 - 1.
```

An even smaller modulus suitable for 32-bit unsigned arithmetic is

```
    M = 239*251; (* = 59989; 239 and 251 are each primes and 3 mod 4 *).
```

More generally you could use iterated cubing (subject to the conjecture it is infeasible to break any bit of the RSA cryptosystem) or in fact any fixed exponent you want (under same conjecture) and in fact almost any fixed integer linear operation on the discrete logarithms of a vector will by the same reasoning be immune to any polynomialtime statistical test, subject to RSA-type conjectures. In particular, we conjecture that the Fibo[a,b,*] generator ought to be immune to polynomialtime statistical tests if the modulus is the product of two suitable large primes... which explains the results of the Marsaglia test battery above.

O Statistic: Statistics

describe

This analysis follows from NR92, pp 610–613. However, the basic definitions are common to all statistics books. The key insight from NR92 (and not original there), is the 2-pass corrected calculation of variance.

To rehearse the equations:

$$\text{mean} = \text{avg} = \overline{x} = \frac{1}{N}\sum_{j=1}^{N} x_j \tag{65}$$

$$\text{avg dev} = \frac{1}{N}\sum_{j=1}^{N} |x_j - \overline{x}| \tag{66}$$

$$\text{var} = \frac{1}{N}\left[\sum_{j=1}^{N}(x_j - \overline{x})^2 - \frac{1}{N}\left[\sum_{j=1}^{N}(x_j - \overline{x})\right]^2\right] \tag{67}$$

$$\text{std dev} = \sigma = \sqrt{\text{var}} \tag{68}$$

$$\text{skew} = \frac{1}{N}\sum_{j=1}^{N}\left[\frac{x_j - \overline{x}}{\sigma}\right]^3 \tag{69}$$

$$\text{kurt} = \left\{\frac{1}{N}\sum_{j=1}^{N}\left[\frac{x_j - \overline{x}}{\sigma}\right]^4\right\} - 3 \tag{70}$$

We also pick up min and max values during the first pass.

ttest

This is Student's t-test. The idea is to find if the means of two samples are the same and how likely that is to be due to chance.

Again, we follow NR92, pg 616, but the formula is common:

$$s_D = \sqrt{\frac{\sum_{i\in A}(x_i - \overline{x_A})^2 + \sum_{i\in B}(x_i - \overline{x_B})^2}{N_A + N_B - 2}\left(\frac{1}{N_A} + \frac{1}{N_B}\right)} \tag{71}$$

$$t = \frac{\overline{x_A} - \overline{x_B}}{s_D} \tag{72}$$

There are a number of ways to compute the probability that this was due to chance. Following NR92, we use:

$$I_x(a,b) = \text{incomplete Beta function} = \frac{B_x(a,b)}{B(a,b)} = \frac{1}{B(a,b)}\int_0^x t^{a-1}(1-t)^{b-1}dt \tag{73}$$

$$\begin{aligned} \nu &= \text{degrees of freedom} && (74)\\ A(t|\nu) &= \text{probability that t could be that small by chance} && (75)\\ &= 1 - I_{\frac{\nu}{\nu_t^2}}(\frac{\nu}{2}, \frac{1}{2}) && (76) \end{aligned}$$

ftest

This is the F-test. The idea is to find if the variances of two samples are the same, and how likely that is to be due to chance.

$$F = \text{var1}/\text{var2} \quad (77)$$

var1 and var2 are chosen so the ratio comes out > 1. There are a number of ways to compute the probability that this was due to chance. Following NR92, we use:

$$\begin{aligned} \nu_1, \nu_2 &= \text{degrees of freedom} && (78)\\ Q(F|\nu_1, \nu_2) &= \text{probability that F could be that small by chance} && (79)\\ &= I_{\frac{\nu_2}{\nu_2 + \nu_1 F}}(\frac{\nu_2}{2}, \frac{\nu_1}{2}) && (80) \end{aligned}$$

chi_sqr1

χ^2 tests whether or not a binned distribution is as expected. NR92 (pg 621, eqn 14.3.1 and pg 221, eqn 6.2.19) gives:

$$\begin{aligned} N_i &= \text{number of items in bin i} && (81)\\ n_i &= \text{number of items expected in bin i} && (82)\\ \chi^2 &= \sum_i \frac{(N_i - n_i)^2}{n_i} && (83)\\ Q(\chi^2|\nu) &= Q(\frac{\nu}{2}, \frac{\chi^2}{2}) = \text{incomplete gamma function P} && (84) \end{aligned}$$

Krey88, pg 1280 says the bins should have ≥ 5 items each. We'll raise an exception if that is not met.

chi_sqr2

This version of χ^2 checks similarlity between two actual collecitons of bins (not just an actual and an expected). NR92, pg 622, give sthe equaiton eqn 14.3.2:

$$\begin{aligned} R_i &= \text{number of items in R's bin i} && (85)\\ S_i &= \text{number of items in S's bin i} && (86)\\ \chi^2 &= \sum_i \frac{(R_i - S_i)^2}{R_i + S_i} && (87)\\ Q(\chi^2|\nu) &= Q(\frac{\nu}{2}, \frac{\chi^2}{2}) = \text{incomplete gamma function P} && (88) \end{aligned}$$

P FourierTransform: Fast Fourier Transforms

[Warren Smith prepared the analysis and modules.]

This is an *in place* FFT routine with array length: $N = 2^n$.

$$a_{output,k} = \sum_{m=0}^{N-1} a_{input,m} e^{(2\pi i \text{direction} mk/N)} \tag{89}$$

where direction $= \pm 1$. You have to do any scaling (by $1/N$ or $1/\sqrt{N}$) or zeroing of a[] by yourself.

Uses $2N \ln N + 2N + O(\ln N)$ [real] multiplications, and $3N \ln N + 2N + O(\ln N)$ [real] additions. (And hopefully the compiler optimizes the subscripting in the inner loop so that only $2N \ln N$ total subscripting ops are needed.)

The test routine shown yields error after 1 forward and one backward use at least 3 decimal places better than slow FT has after only one use!

The basic FFT idea (for N a power of 2) is from J.Cooley & J.Tukey: An algorithm for the Machine Calculation of Complex Fourier Series, MOC 19 (1965) 297-301. (Knuth, Henrici, and Dahlquist/Bjorck also discuss the FFT.) The basic idea is that to calculate the DFT

$$\begin{aligned}
A[k] &= \sum_{j=0}^{N-1} a_j * W^{kj} \text{ for k=0..N-1} && (90)\\
N &= \text{a power of } 2, N > 1 && (91)\\
W &= \text{a principal Nth root of unity} && (92)\\
\text{e.g., } W &= e^{2i\pi/N} && (93)
\end{aligned}$$

We may write

$$\begin{aligned}
A_k &= g_k + h_k * W^k \text{ where} && (94)\\
g_k &= \sum_{0<=2j<N} a_{2j} * (W^2)^{kj} && (95)\\
h_k &= \sum_{0<=2j<N} a_{2j+1} * (W^2)^{kj} && (96)
\end{aligned}$$

are FFTs of half the size (on the even and odd indexed a[]'s, respectively) note that g_k and h_k are periodic in k with period $N/2$, since $W^{N/2} = -1$, so that if $T(N) =$ time to calculate FFT of size N, then $T(N) = 2T(N/2) + O(N)$ and so $T(N) = O(N \ln N)$. The inverse transform is:

$$a_k = \text{Ninverse} \sum_{j=0}^{N-1} a_j \text{Winverse}^{kj} \text{ for k=0..N-1} \tag{97}$$

also an FFT and is calculable by the same method. (Ninverse*N=1. Winverse*W=1.) Cooley-Tukey also works for general highly composite N. A short and elegant implementation is: [Warren E.Ferguson: A simple derivation of glassman's general N FFT, Computers & Math. with Applics. 8 (1982) 401-411].

For an efficient implementation of the FFT, further refinements are desired. These include:

1. Removal of the recursion by "reverse binary permuting" the original data;
2. "In place" implementation with no extra storage requirement;
3. Calculation of the W^k's (k=0,1..) by efficient and stable recurrences (thus avoiding the need for transcendental functions);
4. Possibly use "fast complex multiplication";
5. In specific but common applications, some further savings may be possible. Thus when doing a fast convolution of two arrays of N reals using an FFT, the two FFTs may be accomplished by ONE N point complex FFT, followed by linear time uncombining/termwise multiplication step, followed by a N/2 point reverse FFT. The reverse binary permuting stages may be avoided completely.

The basic symmetry here is that the FFT of a real array is "conjugate even", e.g.

$$A_k = \text{CompConjugate}(A_{N-k}) \text{ if a[0..n-1] is real} \tag{98}$$

R.C.Singleton [e.g. see his algorithm for fast circular convolutions, CACM 12,3 (March 1969) 179; his FFT2 algorithm, CACM 11 (Nov 1968) 773-779; and his article CACM 10 (Oct 1967) 647-654] has suggested a second difference method [which keeps the real and imaginary parts of exp(ick) uncoupled] for evaluating exp(ick):

$$\begin{aligned} e^{ic(k+1)} &= e^{ick} + I_{k+1} && (99)\\ I_{k+1} &= -4\sin^2(c/2)e^{ikc} + I_k && (100)\\ I_0 &= 2i\sin(c/2)e^{-ic/2} && (101) \end{aligned}$$

which is both faster and experimentally far more stable (typically yielding 500 times smaller error for 128 point transform+inverse transform) than the straightforward "multiply by exp(ic)" method. (It's more stable because of the small multiplier $-4\sin^2(c/2)$, as opposed to 1.)

However a better idea which I recently thought of is to use this recurrence:

$$e^{ic(k+1)} = e^{ic(k-1)} + 2i\sin(c)e^{ick} \tag{102}$$

which requires only 2*,2+ per complex exponential (Singleton is 2*,4+; naive method is 4*,2+ and isn't stable) and is also stable since it involves the small multiplier $2i\sin(c)$. (In fact, this method runs faster than Singleton, is easier to program, and even yielded slightly better accuracy in the test program below, too!) Therefore this modification of Singleton is the method I've used.

Incidentally, Press et al. in their NR book use the Singleton algorithm but neglect to credit Singleton!

By a trivial modification of my code, one could generate the sines and cosines by repeated application of the bisection identities:

$$\begin{aligned} \cos(t/2) &= \sqrt{0.5(1.0+\cos(t))} && (103)\\ \sin(t/2) &= 0.5\sin(t)/\cos(t/2) && (104) \end{aligned}$$

starting from the values with $t = \pi$ and $t = \pi/2$ as special cases, thus avoiding trig subroutine calls entirely. OK, I've now done this; now using precomputed table.

The time savings introduced by either of these is small, however. Finally, "radix 8" transforms are experimentally the most efficient, typically 20% faster than "radix 2" routines like this one, although far more complicated; and anyway I suspect the advantage is < 20% in the modern cache-memory system world, since I suspect the radix 2 algorithm has better cache locality. However, that has not been tested.

Also you could take advantage of 1's and 0's to save a little time at the expense of considerably more space.

Another idea which I have chosen not to implement is the fact that two complex numbers may be multiplied in 3 real multiplications: Thus $E + iF = (a + bi) * (c + di)$ may be accomplished via the instructions

$$bpa = b + a; \tag{105}$$
$$bma = b - a; \tag{106}$$
$$E = a * (c + d); \tag{107}$$
$$F = bma * c + E; \tag{108}$$
$$E -= bpa * d; \tag{109}$$

and if bpa and bma are precomputed, this is a 3*,3+ method for a complex multiplication. This idea may be used to reduce (?) the box score from the present $2N \ln N + 2N + O(\ln N)$ mults, $3N \ln N + 2N + O(\ln N)$ adds to $1.5N \ln N + 2N + O(\ln N)$ mults and $3.5N \ln N + 4N + O(\ln N)$ adds. If a floating point multiplication is M times slower than a floating point addition, this idea pays iff $lnN > 4/(M - 1)$. On PDP-11/44 C, however, rough timing has shown that $M = 1.08$ (but with considerable standard deviation. It does about $5 * 10^4$ additions/sec.) so this idea is not worth it unless N is enormous.

Some similar, but worse, ideas have been suggested by Buneman:

```
If c=cos(m),s=sin(m), then precompute
   t1 = (1-c)/s = s/(1+c)= tan(m/2)  and
   t2 = (1+s)/c = c/(1-s).
```

Then $X + iY = (a + bi) * (c + is)$ may be found in 3*,3+ by:

```
if(|t1|<|t2|) then
   X = a-t1*b; Y = b+s*X; X -= t1*Y;
else
   X = b+t2*a; Y = c*X-a; X -= t2*Y;
```

tans may be updated by $\tan(x + y) - \tan(x - y) = 2 * \tan(y)/(1 - (\tan(x) * \tan(y))^2)$, but the extra overhead seems not to be worth it].

There is also a symmetric 3*,5+ (4+ with precomputation) formula for $E + iF = (a + bi) * (c + di)$:

$$E = a * c - b * d \tag{110}$$
$$F = (a + b) * (c + d) - a * c - b * d \tag{111}$$

Other FFT algorithms: Winograd has shown how to design FFTs with N prime (as opposed to the Cooley-Tukey approach which works for N highly composite) that run in $O(N \ln N)$ time and even with only O(N) multiplications; the latter figure is optimal. [S.Winograd: Math. of Comput. 32 (1978) 175-179; Advs in Math 32 (1979) 83-117].

Winograd's approach is based on a theorem that allows him (by a permutation of the original and transformed variables) to express FFTs for N prime in terms of a circular convolution of N-1 elements, plus some additions. He then shows how circular convolutions of k elements (for certain small k) may be computed in a small number of arithmetic operations, (for k=2..6, the number of multiplications Winograd uses is 2,4,5,10,8; for k prime, Winograd shows that a (2k-2)* scheme for CC(k) always exists) and further, how CC(n1) and CC(n2) algorithms may be composed to make a CC(n1*n2) algorithm, IF n1 and n2 are relatively prime, that uses mult(n1)*mult(n2) multiplications. Also, he shows how FFT(n1*n2) may be computed via FFT(n1) and FFT(n2) in mult(n1)*mult(n2) multiplications, IF n1,n2 relatively prime, and also gives methods for FFT(prime power). He gives two appendices containing optimized CC(2..6) and FFT(2..9) algorithms. Winograd's methods don't appear suitable for general N, but if N is specified in advance, they make it possible to do considerable fine tuning at the expense of large algorithm complexity.

Meanwhile, Nussbaumer [H.J.Nussbaumer: Fast Polynomial Transform algorithms for digital convolutions, IEEE Transactions on Audio, Speech, Signal Processing 28,2 (April 1980) 205-215 (this article has many references to other FFT schemes); see also Knuth 2: 503, 652-653] has found another way to do circular convolutions of arrays of (N=a power of 2) reals without any NTTs, FFTs, trig, or complex numbers. His approach is based on viewing circular convolutions as polynomial multiplications modulo certain simple polynomials, factoring the modular polynomials, divide and conquer, chinese remainder thm. His approach uses roughly NlgN *, $N \ln N \ln \ln N$ +, is fairly complicated to program, and requires extra space.

[Nussbaumer & Quandalle: IBM JResDev 22 (1978) 134-144] show how some particularly efficient CC and FFT schemes for N in the range 10-3000 may be constructed; their approach is based on some novel ways to combine efficient small CC schemes that is rather like the NTT (Number theoretic transform) only in rings of polynomials rather than in the integers.

However even the best known arithmetic op count methods only improve on my method by perhaps 30%, and at the cost of considerable complexity. C.H.Papadimitriou [Optimality of the FFT, JACM 26 (1979) 95-102 and its refs] has shown that in some models of computation $O(N \ln N)$ is optimal for the FFT, while Patterson et al have shown a lower bound of $O(N \ln N/ \ln \ln N)$ for integer multiplication on multitape Turing machines, see Knuth 2.

FFTs in a finite field (if the ring ZmodK, called "number theoretic transforms") are discussed in Aho,Hopcoft,Ullman: The Design and Analysis of Computer algorithms, Addison-Wesley 1974. They recommend using W=2, N=a power of 2, do all arithmetic in the ring of integers modulo $2^{N/2}+1$ (in which W is an Nth root of unity, and in which the convolution theorem

$$C_i = A_i B_i \Longleftrightarrow c_i = \sum_{j=0} N - 1 a_i * b_{i-j \bmod N} \tag{112}$$

[Which makes possible the calculation of discrete convolutions in $N \ln N$ time] still holds). [See also R.Agrawal&C.Burrus: NTTs to implement fast digital convolutions, ProcIEEE 63 (1975) 550; articles by H.Nussbaumer on "Fermatrix" and "Mersenne" transforms, IBMJR&D 21 (1976) 282 and 498.]

These FFFFTs are of use in applications where it is desirable to completely eliminate roundoff error and floating point operations, e.g. all-integer convolutions. However, as you can see, NTTs have severe word length and transform length limitations; the need for high precision modular arithmetic can be a major stumbling block. On the other hand, multiplications by W=2 are easy, while modular arithmetic modulo a Fermatrix number is not that hard. Thus using N=16, modulo 65537 arithmetic, W=2 [left shift and modulo], Winverse=32769 [right shift; modular addition correction if inexact], and all numbers in 0..89 allows computation of CC(16) in 16*, many bit shifts and additions.

Rabiner,Schafer,Rader: The Chirp-Z transform and its Applications, BSTJ 48,3 (1969) 1249-1292 show how DFTs (for any N) may be calculated in $N \ln N$ time by using fast convolutions; the method also works for an extension of FFTs (to W=any complex number, not just the principal Nth root of unity):

$$A_k = \sum_{j=0}^{N-1} a_j * W^{k*j} \text{ for k=0..N-1} \tag{113}$$

may be calculated in $N \ln N$ time by a fast convolution by the "Chirp-Z transform" identity

$$A_k = W^{k*k/2} \sum_{j=0}^{N-1} W^{-((j-k)^2)/2} W^{j*j/2} b_j \tag{114}$$

Aho,Stieglitz,Ullman: Evaluating Polynomials at fixed sets of points, SIAMJComp 4,4 (Dec 1975) 533-539, demonstrate that a polynomial and all its derivatives at one point (or equivalently, an origin shift of an Nth degree polynomial) may be calculated in $N \ln N$ time by a fast convolution via the (binomial theorem) identity

$$\sum_{j=0}^{N-1} c_j * (x+q)^j = \sum_{r=0}^{N-1} x^r * d_r / r! \text{ where} \tag{115}$$

$$d_r = \sum_{j=r}^{N-1} c_j * j! * q^{j-r} / (j-r)! \tag{116}$$

Some other applications of FFTs are:

Fast multiplication and division of N digit integers may be done in (roughly) $N \ln N$ time by using fast convolutions followed by a carry step. (See Aho-Hopcoft-Ullman: Design and Analysis of Computer algorithms, for further discussion.)

Base conversion of an N digit number may be done in $N(\ln N)^2$ time by divide and conquer (convert the left and right half of the number recursively, then do a fast multiplication and addition to combine them).

Fast polynomial multiplication and division by fast convolutions in $N \ln N$ time are also discussed in AHU. (This may also be done for Chebyshev series...) Given the N roots of a polynomial, you can find its coefficients (as Chebyshev or as regular) in $N(\ln N)^2$ time by fast polynomial multiplications on a binary tree. On the other hand, you can perform a "root squaring" transformation on a polynomial in (Chebyshev or power form)

$$P(y) = -P(x) * P(-x) \text{ where } y = x^2 \tag{117}$$

in $N \ln N$ time by a fast multiplication, or alternatively can implement a Henrici-Gargantini or Korsak-Pease (or other) simultaneous all root iteration step, in $N(\ln N)^2$ time by a fast multipoint evaluator, see below.

All shifted correlations of vectors (and/or autocorrelations) may be calculated in $N \ln N$ time by fast convolutions; this has application in signal processing, 1D pattern recognition, Electrical engineering.

Fast polynomial multiplication/division/remaindering and a divide and conquering of the Lagrange interpolation formula may be used to do fast Nth degree polynomial interpolation and N-point evaluation, as was shown by Borodin&Moenck [JCompSystSci 1974]. I have extended B&M's results to Chebyshev polynomials and less successfully to other polynomials.

Fast algorithms exist for power-series to continued fraction interconversion; these may also be generalized to Chebyshev series.

Fast polynomial evaluation/interpolation at special point sets (e.g. Z^k for some Z) may be accomplished in $N \ln N$ time by the Chirp-Z and FF transforms; this also carries over to Chebyshev. Thus fast Taylor and Chebyshev series calculations.

Fast composition of Taylor series - $O((N \ln N)^{3/2})$ is also possible, as was discovered by Brent&Kung, via a "block Horner" approach. This may also be extended to Chebyshev.

Fast Elliptic linear PDE solvers (by finite differneces or spectrally): there are many schemes based on FFTs that run in $N \ln N$ time, N=size of output.

A complete list of FFT applications is far too huge to discuss here...