A Portable Floating-Point Environment

David Barnett

ABSTRACT

This paper describes work in progress. In particular, the current syntax of the function calls occurred more as a side-affect of testing the feasibility of implementing this package and the examples which run under it than as a deliberate effort to define a syntactically and semantically clean way of accessing the environment. Any comments you may have on these routines or the contents of this paper would be appreciated. I may be contacted as david@lll-lcc.arpa ({ucbvax!lll-crg, seismo}!lll-lcc!david). Vax and Sun/68881 source code for the routines and examples described herein are also available, as well as a test suite to verify the integrity of an implementation.

This paper describes a set of functions which establish a robust environment for floating-point arithmetic, easing implementation of a great number of problem numerical algorithms and increasing their portability between machines and architectures. Facilities are provided for exploiting underlying arithmetic conforming to or approximating that defined in the *IEEE Standard for Binary Floating-Point Arithmetic* (ANSI/IEEE Std 754-1985). Extensions to Standard requirements allow for efficient exception handling in systems with imprecise interrupts, such as vector and concurrent processors; and for arbitrary magnitude arithmetic without abnormal loss of precision.

These routines are intended to be implementable on, and take maximum advantage of, both Standard conforming and non-conforming architectures. Using the functions described herein, programs can determine what floating-point features are available and access them in a uniform manner. Thus, their use eases the design of portable, numerically sensitive applications. No compiler support is assumed or required.

The package described herein is currently implemented under the UNIX† operating system on the Digital Equipment Corporation VAX, and Sun-3 (with 68881) processors.

December 18, 1987

† UNIX is a trademark of Bell Laboratories.

A Portable Floating-Point Environment

David Barnett

1. Introduction

The functions described in this paper are designed to take advantage of and allow access to features of the *IEEE Standard for Binary Floating-Point Arithmetic* (such as flags, modes, and traps). Several useful extensions to the Standard are also provided. These routines provide a portable bridge between applications software and the underlying floating-point hardware. In addition to allowing access to Standard confirming arithmetic, these functions provide a basis for future hardware floating-point implementations, and are often retrofitable to older, non-confirming architectures.

2. Extensions to the Standard

Two extensions to the IEEE Standard are provided: presubstitution and exponent wrapping.

2.1. Presubstitution

Presubstitution is the ability to specify a value to be used as the result of a specific exceptional operation. The value must be specified in advance of its potential use. There are nine exceptional operations for which a result may be presubstituted: the five IEEE exceptions (underflow, overflow, inexact, division by zero, and invalid), with the invalid exception further subdivided into 0/0, ∞/∞, $\infty-\infty$, $0\times\infty$, and other invalid operations (such as domain errors).

One example of the use of presubstitution is to handle limiting conditions. Prior to computing $x^2/(x^2+7)$, the value 1 may be presubstituted for ∞/∞. If x^2 overflows, the correct value will be returned (as addition of 7 will have no effect on an x^2 large enough to overflow), or the proper limit will be returned if x is in fact infinity. "Presubstitution" was coined by Professor W. Kahan and is discussed in detail in his paper *Presubstitution, and Continued Fractions.*

Functionally, presubstitution is similar to the IEEE notion of a trap handler. The Standard describes a trap handler as being able to determine the type of exception which caused its invocation, as well as the exceptional operands involved. A trap handler should be able to return a value to be used in lieu of the default result. Despite their similarity, presubstitution offers several advantages over post-exception trap handling.

In systems with imprecise interrupts, such as vector and other concurrent architectures, it may not always be possible to determine exactly where an exception occurred. Thus, execution cannot be resumed (without potential side affects) after an exception causes a trap handler to be invoked. In a highly pipelined architecture, speed must be sacrificed in order to allow an exception handler to extract the amount of information it needs, and to resume where processing was previously halted.

For example, consider an architecture in which a floating-point coprocessor is utilized. If floating-point operands are allowed to be external to the coprocessor and exceptions are not signaled until the main processor attempts its first post-exception floating-point operation (as with the Motorolla 68020/68881), an exceptional operand may be overwritten before the main processor becomes aware of the exception. Consider the following program segment:

```
for j = 1 to n do begin
   x[j] := a[j] / b[j];
   a[j] := c[j];
end.
```

After the addresses of *a[j]*, *b[j]* and *x[j]* are computed, the division will be scheduled. While the division is taking place, the copy of *c[j]* to *a[j]* can take place, as well as the incrementing of *j*. If a division exception is signaled, the values of *a[j]* and *j* may have changed from what they were when the division was initiated.

Concurrency would have to be sacrificed (the division waited on) in order to make all variables available to an IEEE trap handler.

Also, consider the problems of a vector architecture in which many identical operations are being performed concurrently. Since any element pair may be responsible for an exception, the exceptional operation must be repeated on each of the vector elements individually, at great expense, to determine which operands were actually responsible for the exception.

Finally, in a highly pipelined architecture, several instructions may be executing simultaneously, and they will not necessarily complete in the order scheduled. The program counter in such as system will refer to the next instruction to be prefetched, rather than to any particular instruction in the execution queue. When an exception occurs, instruction execution can not be resumed at the point of the exception even if it is known (that is, every instruction in the pipeline is tagged with its address). Resuming execution at the exception point would be impossible as side effects of instructions occurring after the exceptional one, but completed before it, cannot always be undone.

Presubstitution solves all these problems as it eliminates the need for an exception to be signaled. In a hardware implementation of presubstitution, the presubstituted value could be stored in a register and accessed by the floating-point hardware just as would be a NaN or other default constant. For example, bits in a control register may be the index of an internal register value to be used as the result of a specific exceptional operation. As a default this index would refer to the IEEE default value, but when presubstitution is specified the prescribed value would be loaded into a register and the index changed to refer to it. At the time of the exception the hardware would react identically whether presubstitution is taking place or not.

Presubstitution also offers syntactic advantages over trap handling performed for the same ends. Unless the value to be presubstituted is a constant, the trap handler must be made aware of that value, most likely through setting of a global variable. In addition to having the same overhead as presubstitution in current software implementations (which actually perform post-exception substitution), this obscures the true objective of the computation.

Additionally, it may appear as if a trap handler could actually compute the value to be substituted in lieu of the result itself, saving the expense of unnecessarily computing this value when it is not needed. This results in a syntactic mess, however. In addition to needing a different trap handler for every computation, the exception handler must have access to all relevant variables from the routine which produced the exception. If these values are copied to global variables, little, if any, time will be saved over actually performing the computation (provided the computation only involves addition, subtraction, multiplication, and division). If the values are global variables, it will not be necessary to copy them; however, recursion will have been precluded.

Many of the syntactic disadvantages of trap handling to substitute valid results for invalid ones can be overcome by avoiding exceptions altogether via "if" tests. If the division a/b could potentially produce an undesired exception, (such as division by 0 or ∞/∞) the test "if b is finite and non-zero..." could be inserted before the division. If the test fails, then an appropriate value would be computed. Unfortunately, if this test appears within a loop while presubstitution to the same ends may be performed outside of the loop, the "if" precludes that loop from taking advantage of a vector architecture. Furthermore, if the value to be presubstituted does not vary between iterations of the loop, the "if" test will be repeated many times, while a value to be presubstituted need only be computed once. If the presubstitution must occur within the loop, then the "if" test may be more efficient than presubstitution. This is considered in more detail in Professor Kahan's paper on this subject.

2.2. Arbitrary Magnitude Arithmetic

A major problem in implementing numeric algorithms on computers is that of underflow/overflow. While underflow is oft neglected, overflow usually aborts computation, or delivers an infinity as the result of an overflowed computation. If the final result of a sequence of computations is indeed too large or small to represent in the chosen precision, an infinity or zero may be a satisfactory result, as numbers as large or small in magnitude as those which would overflow or underflow in conventional floating-point schemes are of little practical application. Unfortunately, an infinity or zero occurring as an intermediate result in a long series of computations, whose final result is representable, will propagate through those computations, potentially leaving no clue as to the true magnitude of the final result.

In traditional and IEEE floating-point arithmetic, a number F is represented as

$$F=\pm f r^{e},$$

where f represents a fixed width, unsigned fraction ($1/r \le f < 1$)); e a fixed width, two's complement, biased exponent; and r the radix of e. Overflow occurs if e is too large to represent, and underflow if e is too small.

Two schemes have recently been proposed to eliminate the problems of underflow and overflow. C. W. Clenshaw and W. J. Olver, in their paper *Level-index Arithmetic Operations*, propose to represent F in the following form:

$$F=\pm e^{e^{\cdot^{f}}},$$

where e is the inverse natural log of 1, $0 \le f < 1$, and a value l corresponds to the number of exponentiations. As l increases, the range of representable numbers is increased, as is the distance between them. Error in F is then relative to the magnitude of l.

S. Matsui and M. Iri propose an alternate representation in *An Overflow/Underflow-Free Floating-Point Representation of Numbers.* In their "level 0" arithmetic, a floating-point number is represented in the conventional sense, with the exception that the widths of f and e are variable. As many available bits as necessary are allotted to e, and the remaining are used to represent f. Thus, for numbers near ± 1, arithmetic is highly accurate (since only one bit is devoted to the exponent and the remaining to the fraction). As numbers move farther from ± 1, bits are shifted from f to e, and arithmetic becomes less accurate. Numbers which would underflow or overflow the level 0 representation are recursively defined as having e itself represented as a level 0 number. The level of a number corresponds to the number of times e is recursively represented by a new floating-point number.

Using either of these schemes greatly eases the job of a programmer implementing an algorithm which may underflow or overflow in conventional floating-point systems by eliminating the potential for underflow or overflow altogether. Unfortunately, the job of the error analyst is correspondingly increased. James W. Demmel points out in *On Error Analysis in Arithmetic with Varying Relative Precision* that arithmetic involving numbers at the extremes of those representable in both schemes are highly inaccurate in comparison to the standard, IEEE adopted format. Furthermore, the accumulated error in a result cannot be easily determined since that error is highly dependent on the number of bits devoted to the mantissa in the Matsui/Iri scheme , and the number of levels of exponentiation in the Clenshaw/Olver implementation.

In order to simplify the error analyst's task, Demmel suggests that a hardware counter be implemented, akin to an IEEE flag, which keeps track of the maximum size obtained by an intermediate result. In Clenshaw/Olver this would be the maximum number of levels of exponentiation, and in Matsui/Iri level 0 arithmetic the maximum number of bits devoted to the exponent. The maximum error inherent in a result could be determined by examining this counter.

The scheme adopted in this package avoids the error analysis problem of the methods utilizing varying precision. Numbers are represented in the traditional IEEE style. If specifically requested, computations experiencing underflow or overflow deliver as their result a number with the same fraction as the true result, and a modified, representable exponent. A counter is incremented on each overflow and decremented on each underflow. The result's exponent is adjusted so that after a sequence of computations like $x=a \times b \times c \times d$, x will be correct and as precise if intermediate underflows and/or overflows occur as if they do not, provided that the underflow/overflow counter is zero at the end of the multiplications.

Traditional error analysis techniques apply to this method, as the width of the mantissa is not varied as is done in the Matsui and Iri scheme. The counter effectively extends the width of the exponent, by allowing it to "wrap around" when it becomes too large or too small (hence, this method is referred to herein as *exponent wrapping*). The counter used in this technique is akin to Demmel's, except that rather than indicating the magnitude of potential error in the fraction, it corresponds to the exact error in the exponent. Accuracy and the ability to perform reliable error analysis are not sacrificed in defense of underflow and overflow.

The primary disadvantage of this scheme is that programmer cognition of an algorithm being susceptible or sensitive to underflow and/or overflow is required. Floating-point operations must be coded to take into account a potentially non-zero wrap count (accumulated underflow/overflow count) being associated with each number. While multiplication and division are simple (wrap counts are added or subtracted, respectively), other operations are more difficult and expensive. Addition and subtraction, for example, require scaling of their operands prior to actually carrying out the arithmetic. Since scaling is affected by the wrap count, addition and subtraction become dependent on the number of bits devoted to the exponent. In order to provide machine independence, a function is included in this package which performs addition on numbers with wrap counts associated with them.

Due to the added expense of dealing with this form of extended arithmetic, it should only be incorporated when needed. Two examples are included later in this paper which demonstrate applications benefiting from this form of arithmetic. One of them runs appreciably faster than an algorithm which solves the same problem and is coded to avoid underflow and overflow in traditional floating-point systems.

This scheme for handling underflow and overflow is also easily implementable in hardware. Underflow and overflow are usually detected when rounding a high precision, internal result to a lesser precision, external one. Instead of generating an exception or substituting an infinity, zero, or denormalized number as the result on detection of underflow or overflow, however, the hardware could use its normal, unexceptional algorithm to deliver the result. The difference between the true exponent and stored one could be channeled to an underflow/overflow accumulator, which could also be accessed and set by applications software. Due to the speed of integer arithmetic, a hardware implementation of this wrapping mode should have no affect on the overall speed of floating-point arithmetic.

3. Environment Access

The code comprising this package is written primarily in 'C', with some assembly language support. While designed to be accessed from 'C' programs, the functions described here should be easily adapted to other languages by appropriately translating the constants defined in the header file. No modification to the code comprising the environment should be necessary.

3.1. Constants

Finite floating-point constants are represented in the normal manner. Positive infinity is specified via the constant **Infinity**, and negative infinity via **NInfinity**. A quit NaN may be introduced via the constant NaN, and a signaling NaN via SNaN. These constants, as well as the declarations of the following functions, are defined in the 'C' header file "fp.h".

3.2. Functions

Unless otherwise specified, all functions return an integer. If a requested feature is unavailable in a specific implementation, or if there was an error in the parameters to a function, the value -1 is returned. Otherwise, the value 0 is returned. A program may take advantage of the -1 return value to adjust itself at run time to the features available in the environment above which it is running.

3.2.1. Initialization and Defaults

This package is initialized via the function call **FPInit()**. **FPInit()** takes no parameters and returns no value. **FPInit()** performs all system dependent initializations and establishes the IEEE Standard default environment.

The current environment may be reset to the default at any point by calling **FPDefaultEnv()**, which is also void and with no parameters.

3.2.2. Flags

Flags are interrogated and set via the function **FPFlag(flag, newVal)**. **FPFlag()** returns an integer: zero if the prescribed flag is not set, and an undefined but positive, non-zero value if the flag is set (-1 is returned if the specified flag was illegal or if it is not supported in the underlying environment). The parameter flag selects the flag being operated on, and is one of the constants **FPFLAG_INV** (for the invalid operation flag), **FPFLAG_OVF** (for the overflow flag), **FPFLAG_DVZ** (for the division by zero flag), **FPFLAG_UNF** (for the underflow flag), or **FPFLAG_INX** (for the inexact flag). A new value for the flag is specified in the **newVal** field. If **newVal** is 0, the flag is cleared, and if it is non-zero the flag is set. The current value of the flag can be preserved (**newVal** ignored) if **FPFLAG_HOLD** is OR'ed into the flag field.

A pseudo-flag is provided for saving and restoring all the flags. This flag is **FPFLAG_ALL**. If it is specified in the flag field, a value is returned which can be later used in the **newVal** field, in conjunction with **FPFLAG_ALL**, in order to set all the flags to their state at the time of the first call. The meaning of the value returned by **FPFlag()** when **FPFLAG_ALL** is specified can not be interpreted by a program unless it is 0, in which case it indicates that no flags are set.

3.2.3. Rounding Modes

Rounding modes are set and interrogated by the function **FPRound(newMode)**. The new mode is specified by **newMode**, and either the previous mode or -1 (in the event of an error or unsupported request) is returned. Valid modes are **FPROUND_NEAR** (the default of round to nearest), **FPROUND_POSINF** (round to positive infinity), **FPROUND_NEGINF** (round to negative infinity), and **FPROUND_ZERO** (round to zero).

An additional mask, **FPROUND_WRAP**, may be OR'ed in with any of the other modes. This mask invokes the exponent wrapping mode. When it is specified, operations which underflow and overflow will produce a result with an exact fraction and legal exponent. The number of underflows and overflows are tabulated in a value which may be requested by calling the function **FPWrapCount(newCount)**. The parameter **newCount** specifies a value to which future exponent underflow and overflow corrections are subtracted and added, and is usually zero.

Since many computations involving extended (wrapped) arithmetic depend on the exponent width, three routines are provided to allow programs utilizing this feature to retain portability. In order to resolve a potential wrap associated with a number, the call **FPResolve(number, wrap)** is provided. The parameter **number** is a floating-point number, and **wrap** is the wrap count associated with it. **FPResolve()** returns a floating-point number equivalent to that represented by the pair **number** and **wrap**. If the quantity is too large or small to represent, an appropriately signed infinity or 0, respectively, is returned. This routine is useful when implementing functions which wish to hide the fact that they are using the **FPROUND_WRAP** mode from their caller.

The other extended arithmetic support routines provided add two potentially wrapped floating-point numbers, and take the square root of one. The call **FPAdd(a, aWrap, b, bWrap)** returns a floating-point number corresponding to "**a + b**", taking into account the wrap experienced by each, represented via the integers **aWrap** and **bWrap**. Any wrap in the result returned by **FPAdd()** is incorporated into the count obtainable by the call **FPWrapCount()**. **FPSqrt(a, aWrap)** likewise returns the square root of the number referred to by **a** (a floating-point number) and its wrap, **aWrap**. Underflow and overflow are also tabulated into the value returned by **FPWrapCount()**.

Consult the accompanying examples for more information on the use of this mode.

3.2.4. Presubstitution

Presubstitution is the ability to specify a value to be used as the result of an exceptional operation. The presubstituted value must be (as the name suggests) specified before an exception occurs. The format of the call is **FPPresubstitute(condition, subVal)**. After a call to **FPPresubstitute()**, the double precision number pointed to by **subVal** is used as the result of any operation which corresponds to condition, which is one of **FPPRE_INFDIV** for ∞/∞, **FPPRE_ZERODIV** for 0/0, **FPPRE_INVMULT** for $0\times\infty$, **FPPRE_INFSUB** for $\infty-\infty$, and **FPPRE_OTHERINV** for any other invalid exception, such as a domain error. For exceptional computations which would raise a flag other then the invalid one, results may be presubstituted using **FPPRE_INX**, **FPPRE_OVF**, **FPPRE_UNF**, or **FPPRE_DVZ** for the inexact, overflow, underflow and division by zero exceptions, respectively. If the pointer **subVal** is **NULL**, then presubstitution is disabled for the specified exception.

FPPresubstitute() returns -1 if presubstitution is not possible on the requested condition (or the condition is illegal), 0 if presubstitution was previously disabled for the specified condition, and 1 if it was previously enabled. In the latter case, **subVal** is set to point to a double precision floating point number which was the previous value being presubstituted when **flag** was raised. If presubstitution is being disabled, **subVal** can not be changed (since the pointer is NULL), so the the previously presubstituted value cannot be determined. However, the currently presubstituted value may be determined prior to disabling presubstitution by making two calls to **FPPresubstitute()**.

Trapping has a higher priority than presubstitution. If trapping is enabled on an exception, than that trap is taken regardless of whether or not the condition resulting in the exception has had a presubstitution value specified for it.

3.2.5. Exception Trapping

In order for a program to catch its own exceptions, the call **FPTrap(flag, newVector, oldVector)** may be used. When **FPTrap()** is called, future events which would raise **flag** instead call the function pointed to by **newVector**. The specified trap handler may not return.

The intention of the trap handler is that it "cleanly" handles fatal errors, and then either calls *exit()* or jumps to a known location within the application's code.

FPTrap() returns -1 if the specified flag is not supported or is illegal, 0 if traps on **flag** were previously disabled, and 1 if they were enabled. In this last case, the pointer **oldVector** is set to point to the previous trap handler on return from **FPTrap()**. If **newVector** is NULL, then trapping on **flag** is disabled.

Trapping on an exception takes priority over presubstitution on a condition which is associated with that exception.

4. Implementation Details

This package is currently implemented for two processors, the VAX running 4.3 BSD UNIX and Sun-3 equipped with a 68881 running SunOS release 3.2 (a 4.2 BSD UNIX derivative). This section briefly details the method of implementation on each, and the constraints inherent in the two architectures.

4.1. VAX

The VAX does not support IEEE Standard floating-point arithmetic, so implementation of this package and an IEEE style environment required using features of the VAX architecture in ways other than that for which they were intended.

While the VAX does not have any floating-point flags, it does have three exceptions, underflow, overflow, and division by zero, which allow the corresponding IEEE flags and exceptions to be simulated. The VAX also does not support infinities or NaNs in the Standard sense, although it does have the feature that certain bit strings, when interpreted as floating-point numbers are *illegal operands*, and their access causes an illegal operand exception. Thus, through use of these illegal operands and their corresponding exception, NaNs, infinities, and infinity arithmetic can be simulated in software via an illegal operand exception handler. Table 1 and Table 2 illustrate the relationship between VAX and IEEE exceptions.

VAX Exception	VAX/UNIX Default	Enhanced Environment VAX Default
Division by Zero	Abort Program	Produce Appropriate NaN or Infinity
Overflow	Abort Program	Produce Infinity
Underflow	Ignored (Flush to Zero)	Flush to Zero
Illegal Operand	Abort Program	Produce Appropriate NaN, Infinity or 0

Table 1. Comparison of Default Handling of VAX Arithmetic Exceptions

IEEE Exception	VAX/UNIX Default	Enhanced Environment VAX Default
Division By Zero	Abort Program	Produce Infinity
Overflow	Abort Program	Produce Infinity
Underflow	Ignore (Flush to Zero)	Flush to Zero
Invalid	Abort Program	Produce NaN
Inexact	Ignore	Ignore (Except on Overflow and Underflow)

Table 2. Comparison of Default Handling of IEEE Arithmetic Exceptions

Using these exceptions, the VAX can be extended to nearly approximate a true IEEE environment; however, several IEEE features can not be emulated. In particular, the VAX representation of floating-point numbers differs from that specified in the Standard. These differences are summarized in **Table 3.**

Format	Exponent	Fraction
VAX Single	8	23
IEEE Single	8	23
VAX Double	8	55
IEEE Double	11	52

Table 3. Differences in Floating Point Types

The VAX does not provide any control over rounding, so the rounding mode set with **FPRound()** only applies to numbers which underflow or overflow. The VAX also automatically disables underflow each time a new function or procedure is called (the state of underflow detection is restored on each return). Since current compilers do not propagate the underflow detection state, a function call is required to explicitly enable underflow detection in each procedure in which underflow may occur. This call, **FPEnableUnderflow()**, takes no parameters and returns no value. It is also supported, as a null function, in other implementations to ease the porting of applications.

Using the techniques employed in the implementation of this package, the VAX environment can be greatly extended from its default. Use of these extensions does not come without costs, however. While the execution time of conventional arithmetic is unaffected by these enhancements, the time required to perform operations on infinities and NaNs is dramatically greater. This is due to the fact that every illegal operand exception results in the operation being attempted having to be simulated in software. Thus, the assignment "x = Infinity" takes 800 times longer than does "x = 1.0". About 40% of this extra time is spent interpreting and simulating the desired operation, and 60% of it in the UNIX kernel processing the exception and return from it.

4.2. Sun-3

The Sun implementation is much simpler and more efficient than the VAX one, owing to the Motorola 68881 coprocessor supporting the IEEE floating-point standard.

There is only one difficulty in implementing this package in its entirety on the 68881. The 68881 architecture requires that the result and one operand of each arithmetic operation be the same 68881 register. One operand may also be anywhere in memory, addressed using the native 68020 addressing modes. The 68881 does not signal arithmetic exceptions to the 68020 until the first post-exception 68881 access. Thus, it is possible that an in memory operand may have been overwritten before an exception is acknowledged. In the event of underflow or overflow, the only way to reliably retrieve one of the operands is from an internal, privileged register (since the 68881 source/destination register is overwritten by the default result). Unfortunately, the current operating system does not allow access to this register, so the exponent wrapping mode for extended range arithmetic is not supported.

5. Examples

Three examples of the use of this environment are provided. The first gives a solution to the problem of evaluating a continued fraction and its derivative. This evaluation may produce an intermediate zero denominator, which in turn produces an infinity. Both infinity arithmetic and presubstitution are employed to determine a solution.

The second and third examples demonstrate the use of the exponent wrapping mode for extended range arithmetic. The first of these provides a solution to an angular momentum problem in which intermediate results have too great a magnitude to be represented in the normal VAX or IEEE single or double formats. This problem was used by Matsui and Iri to demonstrate the utility of their floating-point format. However, using underflow/overflow counting a more accurate result can be achieved on a VAX using the normal double precision format than was obtained by them.

The final example, that of computing binomial probability, was considered by both Matsui/Iri and Clenshaw/Olver. As with the angular momentum problem, it is best solved using the exponent wrapping mode.

5.1. Continued Fraction (Jacobi Form)

Consider a generalized continued fraction of the form

$$f(x)=a_0+\cfrac{b_0}{a_1+x+\cfrac{b_1}{a_2+x+\cfrac{b_2}{\cdots+\cfrac{b_{n-1}}{a_n+x.}}}}$$

There is an iterative solution for $f(x)$ and $f'(x)$, namely

```
f =a_n ;
f'=0;
for m=n-1 downto 0 do begin
    $f' = -(1+f')\frac{b_m}{(x+f)^2}$;
    $f = a_m + \frac{b_m}{f+x}$;
end.
```

Isolating common factors gives

```
f =a_n ;
f'=0;
for m=n-1 downto 0 do begin
    d=x+f ;
    $q=\frac{b_m}{d}$;
    $f' = -(1+f')\frac{q}{d}$;
    f =a_m+q ;
end.
```

In evaluating the continued fraction, a problem arises if some d=0. Computation of f_m would not be affected if the underlying numeric environment supported the generation and propagation of infinities, since, on the iteration when d became 0, $q=\infty$, thus $f=a_m+q=\infty$. On the next iteration of the loop, d will be infinity, so q will be 0 and the infinity removed from the computation. If infinities are not supported, an algorithm coded as that above will often abort even when a legal, finite answer exists due to the potential for intermediate exception.

The derivative is more difficult to evaluate when $d=0$. On the iteration when d vanishes, f' will be infinite, as is f, leading to the invalid assignment $f'=-(1+\infty)\frac{0}{\infty}$ on the following iteration. In this case, the value which should be assigned to f' is $b_m(1+f')/b_{m+}1$. There is an additional hazard in evaluating the derivative. If $1+f'$ is 0 concurrently with d, there will be a different invalid assignment: $f'=-0\times\infty/0$. As with the previous case, the desired value here is $b_m(1+f')/b_{m+1}$.

Fortunately, this problem is not without solution. Using presubstitution, the desired value for an exceptional right hand side of the assignment to f' can be specified in advance, so in the event that an exception does occur the appropriate value will be automatically used. By substituting ∞ for either ∞/∞ or 0/0 and performing the division $(1+f')/d$ prior to the multiplication, the invalid computation of f' will always be revealed as a $0\times\infty$. Thus substituting $b_{m-1}(1+f')/b_m$ for $0\times\infty$ at the *end* of each iteration of the **for** will result in the proper computation being performed on the next iteration. This is illustrated in the following algorithm:

```
presubstitute $\infty$ for $\infty/\infty$;presubstitute $\infty$ for $0/0$;$f=a_n$;
$f'=0$;
for $m=n-1$ downto 0 do begin
    $d=x+f$,
    $d'=1+f'$
    $q=\frac{b_m}{d}$,
    $f'_m=-(q/d)d'$
    $f=a_m+q$,
    presubstitute $b_{m-1}\frac{d'}{b_m}$ for $0\times\infty$;
end.
```

A program implementing this algorithm is given in Listing 1. Consult Professor Kahan for its proof.

```
/* compute f and f' using infinity arithmetic and presubstitution */
void compute(a, b, x, n, f, fprime)
double a[], b[],              /* coefficients */
    x;
int n;                        /* number of coefficients */
double *f, *fprime;                  /* results */
{
  double func,                /* value of f at this iteration */
       deriv,                 /* value of derivative at this iteration */
       d, dprime, q,                 /* common denominator factors */
       to_sub;                /* value which will be presubstituted */
  int j;

  to_sub = Infinity; FPPresubstitute(FPPRE_ZERODIV, &to_sub);
  to_sub = Infinity; FPPresubstitute(FPPRE_INFDIV, &to_sub);

  deriv = 0.0;
  func = a[n];

  for (j = n-1; j >= 0; j--) {
     d = x + func;
     dprime = 1.0 + deriv;
     q = b[j+1] / d;
     /* presubstitution may occur here */
     deriv = -(dprime/d); deriv *= q;
     func = a[j] + q;
     to_sub = b[j] * dprime / b[j+1];
     FPPresubstitute(FPPRE_INVMULT, &to_sub);
  }

  /* store results */
  *f = func;
  *fprime = deriv;
}
```

Listing 1. Solution to Continued Fraction Using Presubstitution and Infinity arithmetic

There is an additional solution to the problem of evaluating a continued fraction and its derivative which does not depend on the underlying arithmetic environment supporting either infinities or presubstitution. This solution, also due to Professor Kahan, avoids division by zero by adding a small perturbation, ε, to the divisor d. This ε should have the property that if $x+f_{m+1}\neq 0$, then $x+f_{m+1}+\varepsilon = x+f_{m+1}$; that is, ε is smaller than any roundoff in the addition of x and f_{m+1}. If $x+f_{m+1}=0$, then d will equal ε, and q will be approximately 1/ε. Thus, 1/ε will function similarly to an infinity. Since the expression q/d appears in the evaluation of f', ε must be large enough so that $1/\varepsilon^2$ is representable. A program to evaluate a continued fraction using the ε method is given in *Listing* 2.

```
/* The following function iteratively computes f and f' avoiding exceptions */
void compute(a, b, x, n, f, fprime)
double a[], b[],          /* coefficients */
    x;
int n;                    /* number of coefficients */
double *f, *fprime;           /* results */
{
  double func,            /* value of f at this iteration */
        deriv,            /* value of derivative at this iteration */
        d, q,             /* common factors */
        epsilon=1.0e-15;  /* to prevent 0 divisors */
  int j;

  deriv = 0.0;
  func = a[n];

  for (j = n-1; j >= 0; j--) {
     /* epsilon has to be added separately so it isn't swamped */
     d = x + func; d += epsilon;
     q = b[j+1] / d;
     deriv = -(1 + deriv)*q/d;
     func = a[j] + q;
  }
  /* store results */
  *f = func;
  *fprime = deriv;
}
```

Listing 2. Solution to Continued Fraction Using ε to Avoid Exceptions

The solution using ε is not as nice as that involving presubstitution, since its validity depends on the chosen coefficients. For certain coefficients, it may be necessary to recalculate epsilon each iteration of the loop. Furthermore, the appropriate value for ε will vary between machines since it depends on the accuracy of arithmetic and the range of numbers. For some coefficients and floating-point implementations, there may be no satisfactory value of ε unless precision is sacrificed.

Table 4 compares the running time of the two programs, on both the VAX and Sun, on the following fraction:

$$f(x)=4-\cfrac{3}{(x-2)-\cfrac{1}{(x-7)+\cfrac{10}{(x-2)-\cfrac{2}{x-3}}}}.$$

Method and Computation		VAX-11/750			Sun-3		
Program	Input	User	System	Total	User	System	Total
Enhanced Environment	Exceptional	18.7	13.4	32.1	0.8	0.3	1.1
	Unexceptional	7.9	5.1	13.0	0.5	0.1	0.6
Epsilon Method	Exceptional	2.3	0.4	2.7	0.5	0.1	0.6
	Unexceptional	2.3	0.2	2.5	0.5	0.1	0.6

Table 4. Running Time of Continued Fraction Solutions

The programs were run with x values of 0,1,2,3,4, and 5. The zero divisor problem is exhibited on inputs x=1,2,3, and 4.

5.2. 6-j Symbols (Racah Symbols)

The 6-j symbols, $\begin{Bmatrix} j_1 & j_2 & j_3 \\ l_1 & l_2 & l_3 \end{Bmatrix}$, are defined by Matsui and Iri as follows:

$$\begin{Bmatrix} j_1 & j_2 & j_3 \\ l_1 & l_2 & l_3 \end{Bmatrix} = \Delta(j_1 j_2 j_3)\Delta(j_1 l_2 l_3)\Delta(l_1 j_2 l_3)\Delta(l_1 l_2 j_3) w \begin{Bmatrix} j_1 & j_2 & j_3 \\ l_1 & l_2 & l_3 \end{Bmatrix},$$

where

$$\Delta(abc) = \sqrt{\frac{(a+b-c)!(a-b+c)!(-a+b+c)!}{(a+b+c+1)!}},$$

$$w \begin{Bmatrix} j_1 & j_2 & j_3 \\ l_1 & l_2 & l_3 \end{Bmatrix} = \sum_z \frac{(-1)^z (z+1)!}{F(z, j_1, j_2, j_3, l_1, l_2, l_3)},$$

and $F(z, j_1, j_2, j_3, l_1, l_2, l_3) = (z-j_1-j_2-j_3)!(z-j_1-l_2-l_3)!(z-l_1-j_2-l_3)!(z-l_1-l_2-j_3)!(j_1+j_2+l_1+l_2-z)! \times (j_2+j_3+l_2+l_3-z)!(j_3+j_1+l_3+l_1-z)!$. According to Matsui and Iri, the "j's and l's are nonnegative half integers such that the three arguments appearing in one and the same $\Delta(abc)$ may satisfy the triangular inequality, and the summation $\sum_z$ is taken for all integers such that the arguments of factorials may be nonnegative."

The problem in evaluating the 6-j symbols is the huge intermediate results generated by the products of factorials. For j and l values greater than 8, intermediate results become too large to store in the VAX or IEEE double format. Final results, however, are of the order of 10^{-3} and 10^{-4}, easily representable.

A program to evaluate the symbols for all values of j and l using the enhancements described in this paper is given in Section 6.1. This program takes advantage of the exponent wrapping mode to keep track of the cumulative exponent underflow/overflow. Hence, expressions with intermediate results out of the normal exponent range may be evaluated without loss of precision (beyond that due to normal rounding).

In order to implement arithmetic with potentially out of range exponents, two pieces of information must be maintained for each real number: the fractional part and the underflow/overflow count. In order to maintain this information, a new type is introduced, called wayout in the listing.

As arithmetic with out of range numbers requires expensive function calls in place of what is normally straightforward expression evaluation with inline multiplication, it significantly slows down the rate at which calculation is done. No gain is had if the numbers involved are in fact wholly representable in the double format. Accordingly, extended arithmetic should be accommodated only if it is absolutely needed to compute results otherwise incalculable.

Using the program given in Section 6.1 to evaluate the 6-j symbols revealed a rather large discrepancy between results obtained using the VAX double format and those claimed by Matsui and Iri using their scheme. In order to determine which set of results was in fact correct, the algorithm was run under VAXIMA, an implementation of MIT's MACSYMA for the VAX. The symbols evaluated at the values selected by Matsui and Iri are revealed by VAXIMA to be rational numbers. VAXIMA postpones converting real expressions, such as $\sqrt{2}$, to finite precision real numbers until explicitly requested to do so. As it turns out, all square roots in the 6-j symbols cancel. The following are the true values of the 6-j symbols:

$$\begin{Bmatrix} 10\ 10\ 10 \\ 10\ 10\ 10 \end{Bmatrix} = -\frac{481673}{165002460}$$

$$\begin{Bmatrix} 20\ 20\ 20 \\ 20\ 20\ 20 \end{Bmatrix} = -\frac{33188637458619}{6598917336119836}$$

$$\begin{Bmatrix} 30\ 30\ 30 \\ 30\ 30\ 30 \end{Bmatrix} = \frac{36082186869033479581}{87954851694828981714124}$$

$$\begin{Bmatrix} 40\ 40\ 40 \\ 40\ 40\ 40 \end{Bmatrix} = \frac{15532984259505189067801665773}{8495829465052598504989585496460}$$

$$\begin{Bmatrix} 50\ 50\ 50 \\ 50\ 50\ 50 \end{Bmatrix} = -\frac{65433637321280755672145420346825568 3}{583512578555569910271819677105299348360}$$

$$\begin{Bmatrix} 60\ 60\ 60 \\ 60\ 60\ 60 \end{Bmatrix} = -\frac{689702489298339102537670065690020673546392459151}{68515625505089320141017758791246464658154267820370 0}$$

Table 5 compares the results obtained using underflow/overflow counting to those obtained by Matsui and Iri.

x	VAXIMA	VAX Double Format	Matsui And Iri
10	-2.919186780609210E-03	-2.919186780609209E-03	-2.919186780609217E-03
20	-5.029406456867957E-03	-5.029406456868011E-03	-5.029406456868*522*E-03
30	4.102353215741345E-04	4.102353215743201E-04	4.102353215969*380*E-04
40	1.828306973839313E-03	1.828306973842854E-03	1.82830697216*5276*E-03
50	-1.121374923626420E-04	-1.121374927247416E-04	-1.12137431660*3310*E-04
60	-1.006635324736411E-03	-1.006635318034354E-03	-1.00663625630*6322*E-03

Table 5. Results Obtained in Evaluation of 6-j Symbols

Figures shown in bold are accurate decimal digits. The italicized figures in the Matsui an Iri results are those which they claimed to be the only inaccurate figures. **Table 6** summarizes the decimal accuracy obtained using the various methods.

Value	VAXIMA	VAX Double Precision	Matsui And Iri (claimed)
10	infinite	14	15 (14)
20	infinite	12	12 (13)
30	infinite	12	10 (13)
40	infinite	11	9 (13)
50	infinite	9	7 (12)
60	infinite	8	6 (12)

Table 6. Decimal Precision in Evaluating 6-j Symbols

A digit is considered correct if the result would be correct to that digit if rounded to it. Noteworthy are the facts that Matsui and Iri were overly optimistic in their error analysis, and that the VAX double format delivers a more accurate result for inputs greater than 10.

5.3. Binomial Probability Distribution

The problem of computing the cumulative binomial probability distribution is considered by both Matsui/Iri and Clenshaw/Olver. The equation for this distribution is

$$I(n,m,p) = \sum_{j=0}^{m} \begin{bmatrix} n \\ j \end{bmatrix} p^j (1-p)^{n-j},$$

where $0 \le p \le 1$ and the integers $n \ge m \ge 0$.

The above equation is equivalent to the following:

$$q = 1 - p,$$
$$t_{n,m} = \binom{n}{m} p^m q^{n-m},$$
$$I(n,m,p) = \sum_{j=0}^{m} t_{n,j}.$$

The t's can be computed iteratively using

$$t_{n,m-1} = \frac{mq}{(n-m-1)p} t_{n,m}.$$

If $t_{n,m-1} < t_{n,m}$, then an optimal approach to this problem would be to sum terms from the largest t to the smallest, as smaller t's may be so small that when added to the cumulative sum of previous t's they may have no affect on the sum. Thus, the $\sum t$ need only be done for the t's which will contribute to the final result.

Unfortunately, application of this strategy requires that $\binom{n}{m}$, p^m, and q^{n-m} all be computed: the first of these expressions is subject to overflow, and the latter two to underflow. As with the 6-j symbols problem, these difficulties may be overcome using exponent wrapping. A program to compute binomial probability distribution using the method outlined above is given in Section 6.2. Table 7 contains an overview of the performance of this program.

n	m	p	result	Execution Time		
				user	system	total
2000	200	0.1	0.5188204006	0.06	0.05	0.11
30000	3000	0.1	0.5048623033	0.8	0.7	1.5

Table 7. Performance of Binomial Probability Distribution Solution

6. Listings

6.1. Functions to Evaluate 6-j Symbols

```
#include "fp.h"

#define MAXFACT 300            /* number of factorials to pre-compute */

/* structure for holding not normally representable floating-point numbers */
typedef struct { double real; int exp; } wayout;

wayout fact[MAXFACT+1]; /* the pre-computed factorials */

void compute_factorials()
{
  register int i;

  FPEnableUnderflow();

  fact[0].real = 1.0;
  for (i=1; i<=MAXFACT; i++) {
     fact[i].real = fact[i-1].real * i;
```

```
        fact[i].exp = fact[i-1].exp + FPWrapCount(0);
    }
}

void delta(a,b,c,result)
int a,b,c;
wayout *result;
{
    wayout numerator,denominator;

    FPEnableUnderflow();
    numerator.real = fact[a+b-c].real*fact[a-b+c].real*fact[-a+b+c].real;
    numerator.exp = fact[a+b-c].exp+fact[a-b+c].exp+ fact[-a+b+c].exp+FPWrapCount(0);
    denominator.real = fact[a+b+c+1].real;
    denominator.exp = fact[a+b+c+1].exp;

    result->real = numerator.real / denominator.real;
    result->exp = numerator.exp - denominator.exp + FPWrapCount(0);

    result->real = FPSqrt(result->real, result->exp);
    result->exp = FPWrapCount(0);
}

void F(z,j1,j2,j3,l1,l2,l3,result)
int z, j1, j2, j3, l1, l2, l3;
wayout *result;
{
    FPEnableUnderflow();
    result->real =  fact[z-j1-j2-j3].real * fact[z-j1-l2-l3].real *
                    fact[z-l1-j2-l3].real * fact[z-l1-l2-j3].real *
                    fact[j1+j2+l1+l2-z].real * fact[j2+j3+l2+l3-z].real *
                    fact[j3+j1+l3+l1-z].real;

    result->exp =   fact[z-j1-j2-j3].exp + fact[z-j1-l2-l3].exp +
                    fact[z-l1-j2-l3].exp + fact[z-l1-l2-j3].exp +
                    fact[j1+j2+l1+l2-z].exp + fact[j2+j3+l2+l3-z].exp +
                    fact[j3+j1+l3+l1-z].exp + FPWrapCount(0);
}

int max3(a,b,c)          /* return maximum of 3 args */
int a, b, c;
{
    a = a > b ? a : b;
    return(a>c ? a : c);
}

int min4(a,b,c,d) /* return minimum of 4 args */
int a, b, c, d;
{
    a = a<b ? a : b;
    c = c<d ? c : d;
    return(a<c ? a : c);
}
```

```
void w(j1,j2,j3,l1,l2,l3,result)
int j1, j2, j3, l1, l2, l3;
wayout *result;
{
   int z, zmin, zmax;
   wayout numerator, denominator, sum, temp;

   FPEnableUnderflow();
   sum.real = 0.0;
   sum.exp = 0;

   zmin = -min4(-j1-j2-j3,-j1-l2-l3,-l1-j2-l3,-l1-l2-j3);
   zmax = max3(j1+j2+l1+l2,j2+j3+l2+l3,j3+j1+l3+l1);

   for (z=zmin; z<=zmax; z++) {
        numerator.real = fact[z+1].real;
        numerator.exp = fact[z+1].exp;
        if (z%2) numerator.real = -numerator.real;
        F(z,j1,j2,j3,l1,l2,l3,&denominator);
        (void) FPWrapCount(0);
        temp.real = numerator.real / denominator.real;
        temp.exp = numerator.exp - denominator.exp + FPWrapCount(0);
        sum.real = FPAdd(sum.real, sum.exp, temp.real, temp.exp);
        sum.exp = FPWrapCount(0);
   }
   result->real = sum.real;
   result->exp = sum.exp;
}

double racah(j1,j2,j3,l1,l2,l3)
int j1,j2,j3,l1,l2,l3;
{
   wayout temp1, temp2, temp3, temp4, temp5;
   wayout result;

   FPEnableUnderflow();
   delta(j1,j2,j3,&temp1);
   delta(j1,l2,l3,&temp2);
   delta(l1,j2,l3,&temp3);
   delta(l1,l2,j3,&temp4);
   w(j1, j2, j3, l1, l2, l3, &temp5);

   result.real = temp1.real * temp2.real * temp3.real * temp4.real * temp5.real;
   result.exp = temp1.exp + temp2.exp + temp3.exp + temp4.exp + temp5.exp + FPWrapCount(0);
   return(FPResolve(result.real,result.exp));
}
```

6.2. Functions to Evaluate Binomial Probability Distribution

```
/* epow(e, k) computes e^k taking into account possible exponent wrapping */
double epow(e, k)
double e;
int k;
{
   register double result;      /* running value of result */
```

```
    register int pow2=1, /* greatest power of two <= k */
             j=1,        /* power currently computed */
             orig_wrap,  /* for preserving entry exponent wrap */
             wrap=0;     /* wrap in exponent of result */

    if (k <= 0) return(1.0);          /* nothing to compute */

    FPEnableUnderflow();              /* for VAX */
    orig_wrap = FPWrapCount(0);           /* preserve and add on exit */

    /* find greatest power of two less than or equal to k */
    while (k-pow2>=pow2) pow2 += pow2;

    /* compute e^k based on the fact that e^k = e^pow2 * e^(k-pow2) */
    result = e;                       /* starting point */
    k -= pow2;                        /* adjusted power of k */
    while (pow2>j) {
        /* compute square */
        result *= result; wrap += wrap;
        j += j;                       /* power computed */
        k += k;
        if (k>=pow2) { k -= pow2; result *= e; }
        wrap += FPWrapCount(0);
    }

    FPWrapCount(wrap + orig_wrap);    /* wrap in result + wrap on entry */
    return(result);
}

/* compute I(n, m, p) */
double binomial2(n, m, p, q)
int n, m;
double p, q;
{
    double p_to_the_m,            /* p^m */
        q_to_the_mn,          /* q^mn */
        m_factorial,              /* m! */
         t,                   /* t(n,m) */
         s,                   /* sum of t's (incorporating roundoff */
         z,                   /* sum of t's */
         c, old_c;            /* roundoff */
    int mn,                       /* min(m, m-n) */
        j;                        /* loop counter */

    FPEnableUnderflow();

    /* take advantage of the fact that nCm = nCn-m to do minimum comp. */
    mn = n-m; if (m<mn) mn=m;

    /* compute m! */
    for(m_factorial=1.0, j=1; j<=mn; j++) m_factorial *= j;
    /* negate the wrap count since this will appear in the denominator */
    FPWrapCount(-FPWrapCount(0));
    p_to_the_m = epow(p, m);
    q_to_the_mn = epow(q, n-m);

    /* compute mn! */
    for (t=1.0, j=1; j<=mn; j++) t *= n+1-j;

    t /= m_factorial;
    t *= q_to_the_mn;
    t *= p_to_the_m;

    t = FPResolve(t, FPWrapCount(0));

    /* t now equals the maximum t */

    /* loop until addition of more terms no longer affects sum */
    for (z = 0.0, c = 0.0, s = t; s>z; m--) {
        z = s;
        /* compute t(n,m) from t(n,m-1) */
        t *= m*q/((n-m+1)*p);
        /* set s to sum of all terms plus previous roundoff */
        s = t+c; s += z;
        /* compute roundoff */
        old_c = c; c = z-s; c += t; c+= old_c;
    }
    return(z);
}

/* compute binomial probability so that t will decrease with m */
double binomial(n, m, p)
int n, m;
double p;
{
    if (m<=(n+1)*p) return(binomial2(n, m, p, 1.0-p));
    else return(1.0-binomial2(n, n-m-1, 1.0-p, p));
}
```

7. Acknowledgements

Development of this package began as an undergraduate project supervised by Professor Paul Hilfinger. Code to handle the exponent wrapping mode is modeled after and in some cases borrowed from that provided by him.

The package in its current form and this paper were developed as a project for Professor William Kahan's *Scientfic Operating Environments* course in the Fall of 1986. Professor Kahan's input has been invaluable in completing this project.

BIBLIOGRAPHY

VAX Architecture Handbook, Digital Equipment Corporation, Maynard, MA, 1981.

IEEE Standard for Binary Floating-Point Arithmetic (ANSI/IEEE Std 754-1985), The Institute of Electrical and Electronic Engineers, Inc., New York, NY, August 12, 1985.

C. W. Clenshaw and F. W. J. Olver, "Level-Index Arithmetic Operations," *SIAM Journal on Numerical Analysis,* vol. 24 No. 2, pp. 470-485, Society for Industrial and Applied Mathematics, April, 1987.

James W. Demmel, "On Error Analysis in Arithmetic with Varying Relative Precision," *Proceddings of the 7th Symposium on Computer Arithmetic,* Como, Italy, May 18-21, 1987.

David Hough, *Floating-Point Programmer's Guide for the Sun Workstation,* Sun Microsystems Inc., 1986.

W. Kahan, *Presubstitution, and Continued Fractions,* Work in progress, April 24, 1987.

Shouichi Matsui and Masao Iri, "An Overflow/Underflow-Free Floating-Point Representation of Numbers," *Journal of Information Processing,* vol. 4, No. 3, pp. 123-133, November, 1981.

Motorola, *MC68881 Floating-Point Coprocessor User's Manual,* Motorola, Inc..