# HPF - High Precision Floating Point Arithmetic in Matlab

John R. D'Errico*

October 16, 2012

*email : woodchips@rochester.rr.com

# Contents

# 1 Introduction

Frequently I see people asking for the ability to do arithmetic in more than the 16 digits that MATLAB provides as a default. The fact is, that wish is not a good enough excuse to write such a tool. Better is for those same individuals to learn good methods of numerical analysis, to learn to do their work without a lazy recourse to high precision. At the same time, there are problems that may simply require a few extra digits, and I cannot honestly judge whether someone really, truly needs a long precision tool.

A better excuse for my writing these tools was for my own education. It was an excuse for me to learn to build a class in MATLAB using the classdef style introduced a few years ago in MATLAB À second goal was to devise some different techniques in numerical methods. And finally, I enjoyed the mental exercise of how one might devise such a tool to be both efficient and accurate.

Many of the concepts used in the HPF class are similar to those used in my VPI class - that used for computations on integers with any number of digits. HPF is different however, since it is a tool for floating point computations, with a fixed, user designated number of digits carried. I'll discuss some of these issues, as well as techniques necessary to get full accuracy for a variety of functions. Other authors might prefer a true fixed point arithmetic, allowing them always to work with a fixed number of decimal digits to the right of the decimal point. While that would be a valuable ability when you are working with monetary values, this seems less flexible to me for scientific computations. Since I don't use MATLAB to balance my checkbook, HPF is a floating point class. [1]

# 2 Design of HPF

First, I'll look at the makeup of a double precision number as defined in MATLAB Ṫhere are 64 bits in each such number. They can be broken down into four distinct component pieces of information.

- The sign of your number, stored in a single bit
- The mantissa, stored in 52 bits
- The sign of the exponent, as a power of 2, again stored in one bit
- The absolute exponent itself, stored in 10 bits

[1] In fact, since HPF is a true decimal numeric class, as long as sufficient digits were specified for the numbers, additions and subtractions on numbers with exactly two decimal digits will always be exact. This is not true for double precision numbers, since decimal fractions are not represented exactly by a double.

All totaled, this amounts to 64 bits of information. Because there are 52 bits in the mantissa, a double can represent any integer as large as $2^{53} - 1$ without error. The dynamic range of a double precision number is essentially $2^{-1023}$ to $2^{1024}$.

While these limits are wide in most computational contexts, the point of writing the HPF class is to remove those limits to every extent possible. As such, I chose to allow the user to specify the number of digits of precision they would carry. HPF would not allow truly variable precision arithmetic, but would allow the user to choose the precision carried in the number.

The goal for HPF is to be a true decimal format, able to store the mantissa as true decimal digits. The pertinent question to resolve is how to store those digits? [2] For example, one might choose to store each digit as a single character, or perhaps an `int8` or `uint8` format. This would allocate a single byte per decimal digit. Even better in terms of storage capability might be to store 2 decimal digits per byte. Since a `uint8` number can store integers in the range 0-255, that would not be a problem in terms of storage. However, for virtually every single computation one would constantly be forced to unpack and then repack the digits into a form that will work for computation. As such, this would be a slow, inefficient storage medium, although it would be fairly nice in terms of memory utilization. Since memory is relatively cheap compared to our desire for fast computation, it makes sense to look for a better compromise, perhaps trading off memory for efficiency. Throughput is what matters most here.

## *m*-Digits

The next logical idea would be to use a double precision vector to store the digits of the mantissa, with each digit as a separate double precision element. Since double precision is the native format used by MATLAB this will be as fast as possible. No unpacking is required for any computations, and we have easy access to every digit of the number. Adds and subtracts will be efficient, since one needs only align the decimal points, then add or subtract the digits as needed, with carries to resolve the result.

This is indeed a very good choice, useful for computations with even several thousand digits. A problem arises however, when we multiply numbers as we will see later. For this reason, the choice was made to store the digits of each number in the form of *m-digits* (Multiple decimal digits) in a single double precision element. Essentially, instead of using base 10 for the storage of any digit, we use base 100, 1000, or even base 1000000.

For example, look at how HPF might store a 20 decimal digit number. Here I'll

[2] In the development of HPF, as a means of comparison, I also wrote a wrapper class for Java.Math.BigDecimal. In fact, this had several serious deficiencies, a major one of which is that the implementation of HPF in MATLAB is considerably faster than are the Java tools. Since pure speed is a big factor here, there was no need to use the Java class at all.

use $\pi$ as the example.

```
>> pie = hpf('pi',[20 0])
pie =
3.1415926535897932384

>> struct(pie)
Warning: Calling STRUCT on an object prevents the object from ...
    hiding its implementation details and should thus be ...
    avoided. Use DISP or DISPLAY
to see the visible public details of an object. See 'help ...
    struct' for more information.
ans =
     NumberOfDigits: [20 0]
        DecimalBase: 4
               Base: 10000
            Numeric: 0
               Sign: 1
           Exponent: 1
             Migits: [3141 5926 5358 9793 2384]
```

By default [3] HPF stores its digits in groups of four digits per double precision element. This is a good compromise between speed of computation and memory efficiency. A nice feature is that many computations need never unpack the $m$-digits into separate decimal digits.

As you can see above, there were a total of 20 decimal digits stored for the mantissa as requested. However, one should never trust the least significant digits in any floating point operation. Therefore, the HPF approach is to offer the ability to carry shadow digits on the mantissa, the value of which are not shown to the user, but are still present for all computations. The user can specify the number of shadow digits to control their confidence in the digits reported. Thus, with 4 shadow digits specified, we see the same digits reported except for the last digit, which is now rounded up to 5 instead of 4.

Here there are 4 unreported shadow digits carried. Arithmetic is done using those digits, but those digits are always left undisplayed, hidden from view. Clearly, one can easily add and subtract these groups of digits as long as the exponents line up. When the decimal points don't line up for addition, the simple solution is to essentially pad one or the other with zeros.

Multiplication of a pair of HPF numbers is as easy, since a convolution does all of the work, and is quite fast in MATLAB (CONV is far better than brute force loops.) However, multiplication brings up another reason why one cannot use small integers to store the mantissa for HPF numbers. As an example, suppose we try to multiply a pair of 10 digit repunit numbers using CONV.

[3] This behavior can be completely controlled by the user, such that the $m$-digits can be any of 1,2,3,4,5, or 6 decimal digits per double element.

```
>> D = repmat(9,1,10)
D =
     9     9     9     9     9     9     9     9     9     9
>> conv(D,D)
ans =
    81   162   243   324   405   486   567   648   729   810   ...
       729   648   567   486   405   324   243   162    81
```

As expected, CONV does the work efficiently, but it generates large intermediate values before carries can convert it back into a list of single digits. The convolution itself should not be allowed to cause an overflow, but had those digits been stored in an `int8` format, even the simple convolution shown here would have overflowed. And since multiple type conversions is a speed reducer, the design decision in HPF was to ignore considerations of memory to store the mantissa, while maximizing the speed of a multiply.

A target of HPF was to work with numbers with 100,000 decimal digits or more. (Even that many digits will be slow in terms of computation on my current hardware. On the other hand, computers do grow faster and bigger with time, so I must allow for larger numbers yet.) How large can such a convolution grow without fear of overflow?

```
>> D = repmat(9,1,100000);
>> max(conv(D,D))
ans =
     8100000

>> log2(ans)
ans =
       22.949
```

This test shows that a multiplication between even a pair of 100,000 digit numbers will never cause a double precision vector to overflow, as doubles can represent integers as large as $2^{53} - 1$. In fact, we can write the maximum possible value of a convolution between a pair of two strings of 9's of length N as $81N$. This suggests that FLINT [4] overflow problems in multiplication between a pair of double precision single digit mantissas will not happen until we are using numbers with as many as $\frac{2^{53}}{81} = 111,199,990,799,272$ digits. On the other hand, had I chosen to store the mantissa as a string of single precision digits, that overflow could happen with numbers with as few as roughly 8,000,000 digits. While that is still a vastly huge number, it is still probable that one day such a number might arise.

If we can store any number up to a huge number of digits as single decimal digits, then why bother with the concept of $m$-digits at all? The problem is that multiplication using a convolution is not linear in the number of digits.

[4] Floating Point INTeger

```
N = [100 1000 10000 100000];
T = zeros(size(N));
for i = 1:numel(N)
  D = repmat(9,[1,N(i)]);
  T(i) = timeit(@() conv(D,D));
end

[N;T]
ans =
       100         1000        10000        1e+05
4.3319e-05   0.00043541      0.01125      0.97662
```

We can see from the above example that the time for a multiply increases roughly with the square of the vector length. Thus a convolution of two vectors of length 100000 takes nearly 100 times as long as a convolution between a pair of vectors of lengths 10000. This logically implies that if we choose to store our digits as $k$ $m$-digits, then a convolution between a pair of such migit strings will be done in roughly $O(k^{-2})$ time compared to the convolution for single digit strings. That can be roughly a 36-1 speedup by stuffing 6 digits into each migit.

That significant gain in speed by using $m$-digits does come at a price. While a convolution between pairs of 9-repunit numbers was shown to be doable using doubles for numbers with trillions of decimal digits, a similar convolution between a pair of numbers stored as 6 $m$-digits may actually cause overflow if we go beyond only a few tens of thousands of digits. To remain assuredly safe of never incurring an overflow, HPF applies the following constraints on the number of digits for a given size migit:

- Numbers stored as 1 $m$-digits are limited in size to $3.6e14$ digits
- Numbers stored as 2 $m$-digits are limited in size to $3.6e12$ digits
- Numbers stored as 3 $m$-digits are limited in size to $3.6e10$ digits
- Numbers stored as 4 $m$-digits are limited in size to $3.6e8$ digits
- Numbers stored as 5 $m$-digits are limited in size to $3.6e6$ digits
- Numbers stored as 6 $m$-digits are limited in size to 36000 digits

A virtue of an HPF number is the ability to store the decimal digits of a number exactly. This too was a design goal. For example, as a double precision number, MATLAB represents the decimal value 1.2 in a binary form. In turn, if we convert for display that value back from the IEEE floating point format back into the exact decimal representation of that number, we will see that most numbers with non-zero fractional part are not represented exactly. This is a frequent cause of consternation for novice users of MATLAB (or any floating point environment for that matter.) We can see the difference in how we choose to create the HPF number 1.2.

```
>> DefaultDecimalBase 1
>> hpf(1.2,[53 0])
ans =
    1.19999999999999995559107901499373838305473327636718 75

>> X = hpf('1.2',[53 0])
X =
    1.2

>> struct(X)
ans =
    NumberOfDigits: [53 0]
       DecimalBase: 1
              Base: 10
           Numeric: 0
              Sign: 1
          Exponent: 1
            Migits: [1 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ...
                0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ...
                0 0 0 0 0 0 0]
```

Specification of X in a string form to HPF allows it to parse the digits directly into HPF form, without the intermediate IEEE754 approximation.

The exponent in an HPF number is stored as a double precision number itself. This allows exponents as large as $2^{53} - 1$, i.e., 9007199254740991. Again, this seems a reasonable limit. While I might have chosen to make the exponent an int64 number, which would have allowed exponents in the range $[-9223372036854775808, 9223372036854775807]$, at the current time, int64 arithmetic is just beginning to become available in the newest MATLAB releases. Use of int64 for the exponent would then limit the use of HPF to the very most recent MATLAB releases.

Taking apart the HPF number above, we can see that it stores 1.2 in a scientific form as 0.12000 times $10^1$.

Finally, numbers which are `Inf` or `NaN` in MATLAB are flagged as such in the 'Numeric' field of the corresponding internal structure. All of the above factors (and many more that I've forgotten or glossed over) were considered in the design of the HPF class in MATLAB.

## 2.1 Defaults on the Number of Digits and the $m$-digit Base

A feature of MATLAB is all numbers default to double precision. Thus if we type `N = 17;` in MATLAB N will be of class double. In the HPF class, we can create an HPF number using the constructor for HPF. When you first install HPF, there are several defaults employed automatically.

```
>> p = hpf('pi')
p =
    3.14159265358979323846264338327950288419716939937510582097494459
>> p.NumberOfDigits
ans =
    64     4
```

By default, an HPF number carries 68 decimal digits, 4 of which are shadowed from view. This idea of shadow digits, i.e., spare or hidden digits is used in MATLAB itself. We can prove this by some simple tests done in MATLAB Ṡet the format to be as long as possible.

```
>> format long g
>> pi
ans =
           3.14159265358979

>> log10(eps)
ans =
          -15.653559774527

>> pi == (pi + eps(pi))
ans =
     0

>> pi + eps(pi)
ans =
           3.14159265358979
```

In effect, we see that while MATLAB reports at most only 15 decimal digits on the value of pi, we know that matlab carries 52 binary bits in the mantissa. For example, MATLAB knows that the numbers $\pi$ and $\pi + \text{eps}(\pi)$ are distinct, yet it shows the same value for both of those numbers. In effect, MATLAB hides away a binary bit or two that it never reports. Of course you can extract those bits, but that takes some effort.

The HPF tools use these shadow digits to as best as possible always return the correct values for the full number of digits requested, because an extra digit or so are always carried in all operations. And since the user can control how much spare precision is carried, this design allows the user complete control of their computations. For example, we can define $\pi$ as an HPF number, reporting 15 decimal digits, with two spare digits hidden away. Those shadowed digits are easily enough uncovered if you wish, simply by telling HPF to report all of the digits it carries.

```
>> DefaultDecimalBase 1
>> pie = hpf('pi',[15 2])
pie =
3.14159265358979

>> mantissa(pie)
ans =
     3  1  4  1  5  9  2  6  5  3  5  8  9  7  9  3  2

>> augmentdigits(pie,[17 0])
ans =
3.1415926535897932
```

Suppose that we desired to work in 105 digit precision, of which the last 5 digits are shadowed from view? This too would be easily accomplished.

A goal of mine in the design was to not force the user to specify the number of digits to be carried every time they create an HPF number. For this purpose, a function named DefaultNumberOfDigits was written. As a function with no arguments, it returns the current default value for the number of digits to be used in an HPF number.

```
>> DefaultNumberOfDigits
ans =
    64     4
```

We can change the default as a command in MATLAB Ṙom that point onwards, a new HPF number will have the specified number of digits, although existing HPF numbers will naturally be untouched. And of course, you can always specify the number of digits to be carried explicitly in the call to HPF. The permanent setting used by DefaultNumberOfDigits is stored as a preference, using the getprefs and setprefs tools in MATLAB

```
>> DefaultNumberOfDigits 30 3
>> DefaultNumberOfDigits
ans =
    30     3
>> hpf('e')
ans =
    2.71828182845904523536028747135
```

That default value will be remembered as a preference, carrying over even to your next MATLAB session, until a new default is specified. [5] One can always override the current default by explicitly specifying the number of digits in the

[5] Within a single MATLAB session, persistent variables are employed to retain the default number of digits information. This avoids the need for repeated access using getpref, which would be excessively slow since it requires MATLAB to read a file off disk for every access.

call to the HPF constructor.

Another useful decision was to allow the user to specify a temporary change to the default number of digits, to be valid only for the current MATLAB session. If you restart MATLAB, or do a "clear functions" command, the default will return to the previous default value.

```
>> DefaultNumberOfDigits 64 2
>> DefaultNumberOfDigits
ans =
     64     2

>> DefaultNumberOfDigits 100 5 session
>> DefaultNumberOfDigits
ans =
    100     5

>> Clear functions
>> DefaultNumberOfDigits
ans =
     64     2
```

The migit base used by HPF is controlled by the function DefaultDecimalBase. Like DefaultNumberOfDigits, it can be used in a command or functional form. Thus, the initial default will indicate the use of 4-*m*-digits for all numbers.

```
>> DefaultDecimalBase
ans =
      4

>> pie = hpf('pi',[20 4])
pie =
3.1415926535897932385

>> pie.Migits
ans =
         3141        5926        5358        9793        2384 ...
                     6264
```

Now, change that default for the decimal base to presume 6-*m*-digits. The actual digits are the same, merely packaged more densely.

```
>> DefaultDecimalBase 6
>> pie = hpf('pi',[20 4])
pie =
3.1415926535897932384
>> pie.Migits
ans =
       314159      265358      979323      846264
```

Finally, we must accept that an HPF number employing k-*m*-digits can store only numbers with an integer multiple of k as the total number of digits. This may force HPF to keep a few extra shadow digits hidden away. In this next example, instead of the total number of 25 digits, 5 of which were to be shadow digits, HPF was forced to employ 10 shadow digits in that total of 30 digits.

```
>> pie = hpf('pi',[20 5])
pie =
3.1415926535897932384

>> pie.Migits
ans =
       314159      265358      979323      846264      338327
```

It can be useful for all users to understand how the decimal base interacts with the number of digits plus the shadow digits, although the decimal base can be largely ignored for almost all computations. Extra shadow digits will simply be hidden away from view. On the other hand for the user who simply insists on working in a total of EXACTLY 13 decimal digits of precision, it too can be done easily.

```
>> DefaultDecimalBase 1
>> DefaultNumberOfDigits 13 0
```

If however, you know that you will never work in more than 30000 or so digits of accuracy, then you will get maximum computational performance by setting the decimal base to 6. Your performance on multiplications will be roughly twice as fast as it would be for a decimal base that uses 4-*m*-digits.

# 3 Arithmetic

## 3.1 Addition, Carries and Tens Complement Arithmetic

One of the first problems to solve in writing a code like HPF is addition. How might we add two numbers? The HPF solution is to resolve any exponent differences by a shift, appending zeros where necessary. Then just add the pairs of digits. (I'll talk about what to do for negative numbers next.) For example, 926.23 + 97.35 yields a vectorized add in MATLAB of:

```
>> [9 2 6 2 3] + [0 9 7 3 5]
ans =
           9    11    13     5     8
```

There are two elements in this result that are not in the set {0 1 2 3 4 5 6 7 8 9}. Resolving that pair of carries yields the "digit" vector [10 2 3 5 8]. In turn the first digit must now be carried, for a final result of 1023.58. An important thing to recognize is that carries can be done in a simple vectorized form using the find function. Thus one need not write an explicit loop, testing each and every element of the number for a carry.

Adding a negative number to a positive is equivalent to subtracting the digits, then performing a tens complement operation. Consider this strange way of writing zero, as $0 = -1 \times 1000 + 9 \times 100 + 9 \times 10 + 10 \times 1$. This idea has the virtue of easily allowing us to add numbers with different signs. For example, how might we accomplish $12.34 - 23.60$?

Convert this into the sum of two 5 digit strings of numbers, with -23.60 made into its tens complement form as effectively $0 - 23.60$. Thus,

```
>> [0 1 2 3 4] + [-1 7 6 3 10]
ans =
    -1     8     8     6    14
```

Resolve these carries using arithmetic modulo 10, and we get a new "digit" string [-1 8 8 7 4]. Now, since the first digit is negative, we undo the tens complement operation, to interpret this as $0 - 1126$. Finally, insert a decimal point back where it belongs, to find that $12.34 - 23.60 = -11.26$.

In general, the same ideas apply for addition and subtraction operations on k-*m*-digits, for $k > 1$.

## 3.2 Multiplication

The simple approach to a multiply is that taught us in elementary school. For example, how might we multiply the pair of numbers 12.3 and 45.67?

```
>> format long g
>> 12.3*45.67
ans =
                561.741
```

The trick to doing this by hand is to implicitly expand the numbers into a sum of terms, thus $12.3 = 1 \times 10 + 2 \times 1 + 3 \times 0.1$. Likewise, we can expand 45.67 into a sum of terms multiplied by powers of 10. Multiplication of the pair of numbers is now simply a matter of multiplication of all combinations of terms, and summing the results, carefully combining the terms with like powers of 10. At the end, if any carries are necessary, then perform those operations. A problem is efficiency. Those explicit loops in MATLAB would be terribly slow. Far better is to recognize that conv does much of the work in this operation for us.

```
>> conv([1 2 3],[4 5 6 7])
ans =
     4    13    28    34    32    21
```

See that there are many digits that will require carries.

```
>> ind = find(result > 9)
ind =
     2     3     4     5     6
>> remainder = mod(result(ind),10)
remainder =
     3     8     4     2     1
>> carry = (result(ind) - remainder)/10
carry =
     1     2     3     3     2
>> result(ind) = remainder;
>> result(ind - 1) = result(ind-1) + carry
result =
     5     5    11     7     4     1
```

Inspection of this result will find only one more digit that requires a carry.

```
>> ind = ind(result(ind) > 9)
ind =
     3
>> remainder = mod(result(ind),10)
remainder =
     1
>> carry = (result(ind) - remainder)/10
carry =
     1
>> result(ind) = remainder;
>> result(ind - 1) = result(ind-1) + carry
result =
     5     6     1     7     4     1
```

As you can see, the multiplication requires no more than a call to conv, followed by a while loop applied to a vectorized set of carries. In the end, resolve where the decimal point lies, and the sign of the result. Thus we can see that floating point multiplies are quite easy to code in a language like MATLAB, especially since we have access to a fast and efficient tool like conv.

In fact, it is efficiency of multiplication that drove my choice of how to store the digits of an HPF number as individual decimal digits, each of which is a double. This would seem to be a memory intensive solution. Why not instead store numbers that must always be integers from the set {0,1,2,3,4,5,6,7,8,9} as one byte integers? MATLAB can indeed operate on `uint8` numbers, or `int8`, if you will allow negative integers. The problem is seen when I try to perform a multiply like

99999*99999.

```
>> N = \texttt{uint8}([9 9 9 9 9]);
>> conv(N,N)
Warning: CONV2 on values of class UINT8 is obsolete.
         Use CONV2(DOUBLE(A),DOUBLE(B)) or ...
            CONV2(SINGLE(A),SINGLE(B)) instead.
> In \texttt{uint8}.conv2 at 11
  In conv at 39
ans =
    81   162   243   324   405   324   243   162    81
```

As you can see, we will have an overflow, since `uint8` is restricted to integers from 0 to 255. MATLAB recognizes the problem, and automatically converts the numbers to doubles anyway. The flaw is that frequent class conversions like this back and forth would be time consuming, and likely sources of bugs in the code. All of this is resolved by use of k-*m*-digits. In the example above, we considered how to multiply numbers in a 1-migit form. As HPF numbers with a decimal base that uses 4-*m*-digits, it gets easier.

```
>> a = hpf('12.3',[8,0])
a =
12.3

>> b = hpf('45.67',[8,0])
b =
45.67

>> struct(a)
ans =
    NumberOfDigits: [8 0]
       DecimalBase: 4
              Base: 10000
           Numeric: 0
              Sign: 1
          Exponent: 2
            Migits: [1230 0]

>> struct(b)
ans =
    NumberOfDigits: [8 0]
       DecimalBase: 4
              Base: 10000
           Numeric: 0
              Sign: 1
          Exponent: 2
            Migits: [4567 0]
```

In fact, we see that the entire multiplication is done simply by $1230 * 4567 = 5617410$. HPF does all of the bookkeeping, carrying along the exponents properly

of course.

```
>> a*b
ans =
561.74100

>> struct(a*b)
ans =
    NumberOfDigits: [8 0]
       DecimalBase: 4
              Base: 10000
           Numeric: 0
              Sign: 1
          Exponent: 3
            Migits: [5617 4100]
```

## 3.3 Division

As with multiplication, we could simply perform long division for any divides that arise, using the methods taught us all in grade school. However, this would be extremely computationally intensive, especially for numbers with many thousands of digits. At first glance, deconvolution might be an alternative. After all, conv yields a great speed boost for multiplies. We can see however that deconv does not work as well as we might like from only a simple test.

```
>> deconv([5 6 1 7 4 1],[4 5 6 7])
ans =
         1.25   -0.0625   -1.546875
```

Since the answer should have been the vector [1 2 3], we have little choice but to go on to greener pastures. A nice trick for division is to use Newton's method, applied to the function $f(X) = \frac{1}{X} - D$, to compute the inverse of the number D.

$$X_{n+1} = X_n - \frac{f(X_n}{f'(X_n)} = X_n(2 - DX_n) \tag{3.1}$$

This iteration will be quadratically convergent for the reciprocal of the number D, so if we start out with 16 significant digits in the inverse of D (converting D to a double gives us that as a start) then after one iteration, the inverse will be correct to roughly 32 digits, then 64 digits, then 128 digits, etc.

Finally, an actual division $\frac{N}{D}$ can be accomplished by the simple expedient of multiplying N by the reciprocal of D.

The scheme above is best when we must divide my a general HPF number D. We can do better of course when the divisor D is a small integer. For example,

division by 5 is equivalent to multiplication by 2, coupled with a divide by 10. Since a divide by 10 is simply an exponent shift for a HPF number, the divide by 5 requires essentially no more than a simple multiply by 2.

Do we gain by the use of $k$ $m$-digits in division? Yes. For example consider the following division between a pair of 10000 digit numbers.

```
>> DefaultDecimalBase 1
>> e = hpf('e',10000);
>> pie = hpf('pi',[10000]);

>> timeit(@() e./pie)
ans =
             0.629758637429

>> DefaultDecimalBase 6
>> e = hpf('e',10000);
>> pie = hpf('pi',[10000]);

>> timeit(@() e./pie)
ans =
             0.119062845429
```

Here it appears that the division using 6-migit numbers was nearly 6 times faster than the same problem using single digit numbers. This makes sense, since division as implemented in HPF uses both addition and multiplication operations. Addition will of course be linear in the number of digits.

# 4 The Elementary Functions

There are four groups of elementary functions :

1. **Reciprocal-Square Root** : $1/x,\ \sqrt{x},\ 1/\sqrt{x}$
2. **Exponential** : $e^x, \log_e x,\ 2^x,\ \log_2 x$
3. **Circular** : $\sin x,\ \sin^{-1} x, \cos x,\ \cos^{-1} x,\ \tan x,\ \tan^{-1} x$
4. **Hyperbolic** : $\sinh x,\ \sinh^{-1} x,\ \cosh x,\ \cosh^{-1} x,\ \tanh x,\ \tanh^{-1} x$

## 4.1 The Exponential Function

Computation of the exponential [6] of a number $x$ employs several artifices to reduce the difficulty of the problem. The simple Taylor series expansion for the ex-

[6] A classic reference on the evaluation of functions is *Computer Approximations*, John F. Hart, et al. While much of the text is devoted to listing explicit approximations, this text also does a

ponential function is a good start, but this series will not converge rapidly enough for our purposes for all values of x.

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!} = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \cdots \tag{4.1}$$

See that for large values of $|x|$, each term in the Taylor series sum will grow in magnitude until that point where n exceeds the value of x. Our goal is to minimize the number of terms taken in the series. This aids in speed of course, but it also reduces error losses. Every arithmetic operation adds a tiny amount of round-off error. So any solution that reduces the number of terms taken in this series will not hurt our cause.

The first artifice is a range reduction. We might subtract off any integer from x, then multiply (or divide) our final result by the corresponding integer powers of $e = 2.71828\ldots$. We use the simple identity that $e^{x+y} = e^x e^y$. Of course, this forces us to store the value of e, a useful number that is tabulated for HPF out to 100,000 decimal digits. Essentially, the integer part taken is $round(x)$, which yields a fractional part in the interval $[-0.5, 0.5)$. [7]

Computation of $e^N$, where N is an integer is itself achieved quite easily by a binary representation of N as the sum of powers of 2. [8] At the same time, the binary expansion for an exponent is always a good solution that is easy to generate, and nearly optimal.) Thus, we would generate $e^{12.34567}$ by splitting the problem into pieces.

$$e^{12.34567} = e^4 e^8 e^{0.34567} \tag{4.2}$$

Those integer powers of e are formed quickly by repeated squaring. Negative integer powers are most easily done by computing the exponential of the absolute value, then by a single divide at the end. This is done since divides are relatively expensive for HPF numbers compared to multiplies.

We are now reduced to the problem of computing $\exp(x)$, for $|x| \leq 0.5$. Here we see that the Taylor series for $x$ will be fairly rapidly convergent. In fact, the $n = 100$ term in that series will be a very small number, on the order of $10^{-188}$.

splendid job of describing techniques like range reduction that will improve convergence of series and other approximations.

[7] I might also have written this code by performing an immediate divide by $\log(10)$. Taking the floor of that result would allow us to reduce the problem to effectively computing $\exp(z)$, where z lies in the interval $[0, 1)$. Then we would merely shift the exponent of the result by an appropriate number of powers of 10, as given by the divide.

[8] The binary expansion is not always the maximally efficient way to compute an integer exponential, as shown in Project Euler problem 122. http://projecteuler.net/index.php?section=problems&id=122

```
>> 0.5^100/factorial(100)
ans =
     8.45272575844283e-189
```

As importantly, see that the 200th term in this sequence of terms will be smaller by roughly $\frac{1}{200}$, since each term in the Taylor series for $\exp(x)$ is related simply to the previous term. $T_n = T_{n-1}\frac{x}{n}$. If our goal is 1000 digits of precision, we could in fact predict how many terms to take, by solving the relation $10^{-1000} = \frac{x^n}{n!}$. Taking a base 10 log and the use of Stirling's approximation for the factorial will suffice nicely here, coupled with a binary search.

```
>> -1000*log(10)
ans =
          -2302.58509299405

>> fun = @(n,x) log(x)*n - gammaln(n+1);

>> fun(403,0.5)
ans =
          -2297.82836182677

>> fun(404,0.5)
ans =
          -2304.52292388529
```

So if $x$ were as large as 0.5, we must take at least 404 terms in the series to assure us of 1000 correct digits in the result. However, what if $x$ were smaller? What if we could reduce $x$ to be on the order of 0.1?

```
>> fun(324,0.1)
ans =
          -2298.80803673945
>> fun(325,0.1)
ans =
          -2306.89444701477
```

Effectively, we see that the Taylor series converges 25% faster for $x = 0.1$ than it does for $x = 0.5$. How does this give us a benefit? It does because a high precision value for $\ln(10)$ is stored in the HPF toolbox. It is necessary for efficient computation of $\ln(x)$. In fact, 500,000 digits of this number are stored for quick access, just as with the values of $e$ and $\pi$.

```
>> x = hpf('ln10',100)
x =
    2.30258509299404568401799145468436420760110148862877297603332
9009675726096773524802359972050895982 98
>>
```

One might ask, how does this benefit computation of the exponential function for some value of $|x| < 0.5$? The trick is best shown by an example. Suppose one wished to compute the value of $e^{123.456789}$? The scheme suggested so far would have us compute this number as:

$$e^{123.456789} = e^1 e^2 e^8 e^{16} e^{32} e^{64} e^{0.456789} \tag{4.3}$$

A better scheme will have us find this reduction:

$$123.456789 - 28log(10) = 58.984406 = 59 - 0.015594. \tag{4.4}$$

That will obviously yield a more rapidly convergent Taylor series.

$$e^{123.456789} = e^1 e^2 e^8 e^{16} e^{32} e^{-0.015594...} \tag{4.5}$$

By the same logic as above, suppose we wished to compute the above result to 100 digits of accuracy? The Taylor series for $e^{0.456789}$ will require 31 terms to yield 100+ digits of precision, yet only 17 terms are needed for 100 digits of precision in the series for $e^{-0.015594}$. A second source of gain comes from never needing to compute $e^{64} = e^{32}e^{32}$.

There is one more golden nugget to be found in the Taylor series for $e^x$. A quick test on the amount of time required for a 100 digit divide in HPF shows it to be roughly 20 times as slow as a multiply of the same size.

```
>> A = hpf(rand(),100)
A =
0.097540404999409524577913543907925486564636230468 75
>> B = hpf(rand(),100)
B =
0.278498218867048397129337899968959391117095947265625
>> timeit(@() A.*B)
ans =
             0.001328207634
>> timeit(@() A./B)
ans =
             0.028584147634
```

So, can we minimize the need to perform divides wherever possible? Look at the Taylor series again.

$$1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \cdots \tag{4.6}$$

Suppose we sum the series backwards? Start at the last term. We already know how to find the last necessary term to achieve a given tolerance on the result, since for $|x| \leq 0.5$, we are assured that the last term taken in the series must dominate

all terms that could possibly succeed it. A good way to control roundoff error on series with terms that decrease monotonically in magnitude is to sum them in reverse order. So this might be a good idea in any event.

$$\frac{x^n}{n!} + \frac{x^{n-1}}{(n-1)!} + \cdots + \frac{x^2}{2!} + x + 1 \tag{4.7}$$

If we compute those factorials for each term, and divide by that value, this will get expensive to compute of course. But, suppose we get tricky here? Try factoring out $n!$, and then evaluate this series using a Horner scheme,

$$\frac{1}{n!}[(\cdots(((x+n)x+(n-1))x+(n-2))x+\cdots 1)x+1] \tag{4.8}$$

Interestingly, you should see that this form requires only a single divide, performed at the very end. Written in MATLAB as a loop, the series approximation for a given number of terms is simply written.

```
Fz = hpf('1',NDig);
Fact = Fz;
for m = mterms:-1:1
   Fact = Fact*m;
   Fz = z*Fz + Fact;
end
Fz = Fz./Fact;
```

A similar scheme will be used for other elementary functions in the HPF class.

## 4.2 Natural Logarithms

Efficient computation of the (natural) logarithm in the HPF class uses a few tricks too. Again, we can start by looking at the Taylor series for $\log(x)$. [9]

The first artifice one will use is a range reduction. Given the floating point format of a number in HPF form, we have $X = M10^E$, for mantissa M and exponent E. Here M is effectively a number that lies in the half open interval [1,10).

$$\log(X) = E\log(10) + \log(M) \tag{4.9}$$

Recall that HPF stores 500,000 digits of $\log(10)$ for quick access, so computation of $\log(x)$ reduces to little more than an accurate computation of $\log(M)$.

One trick that might work well is Newton's method, applied to $e^x$. Since even the exponential function is computed using a series, repeated calls will be slower

[9] Some will prefer to call this function $ln(x)$. Since MATLAB uses $\log(x)$ for that purpose, I will remain consistent.

than I would like. There are in fact, several direct series we might consider using to compute the log. The first, valid only for $0 < x \leq 1$, will generate $\log(x)$.

$$\log(1+x) = \sum_{n=1}^{\infty} \frac{-1^{n+1}}{n} x^n = x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \cdots \tag{4.10}$$

You should see that this series will be quite slowly convergent when $|x|$ is anywhere near 1. A far better series involves a transformation. Thus, for $x > 0$,

$$y = \frac{x-1}{x+1} \tag{4.11}$$

$$\log(x) = 2y \sum_{n=0}^{\infty} \frac{y^{2n}}{2n+1} = 2y(1 + \frac{y^2}{3} + \frac{y^4}{5} + \frac{y^6}{7} + \cdots) \tag{4.12}$$

This series is more rapidly convergent than the first, but for x near 10, y will still be uncomfortably near 1. Can we do better? Perhaps so. Suppose we already have a good approximation to $\log(M)$, the mantissa? Call that approximation L. Now, use a very simple but very accurate scheme for an essential range reduction.

$$\log(M) = L + \log(Me^{-L}) \tag{4.13}$$

The nice thing is we can use the double precision arithmetic in MATLAB to gain a very good approximation L. For example, compute an approximation to $\log(3)$, accurate to 200 significant decimal digits.

```
DefaultNumberOfDigits 200
X = hpf('3');
L = hpf(log(double(X)));

x = X.*exp(-L)
x =
    1.00000000000000013133163257501601628655879303568537 9
75655729491724963154599226214018313928998760148539180 7
74525523149222971652996968233126698089329360761859355 9
25166162840626608716264399326665753383 14

y = (x-1)./(x+1)
y =
    0.00000000000000006566581628750800383127996781309096 7
84221324144492722045089807704663810536945064434595333 9
38516112837713201129506798644326073174879385069351623 0
79009298721215187506525753577497772578274913112887005 8
407

L1 = L + 2.*y
ysq = y.*y;
```

```
L2 = L + 2.*y.*(1 + ysq/3)
L3 = L + 2.*y.*((ysq./5 + hpf(1)./3).*ysq  +1)
L4 = L + 2.*y.*(((ysq./7 + hpf(1)./5).*ysq + hpf(1)./3).*ysq  +1)
L5 = L + 2.*y.*((((ysq./9 + hph(1)./7).*ysq + hpf(1)./5).*ysq + ...
    \cdots
     hpf(1)./3).*ysq  +1)
L6 = L + 2.*y.*(((((ysq./11 + hpf(1)./9).*ysq + hph(1)./7).*ysq ...
    + \cdots
    hpf(1)./5).*ysq + hpf(1)./3).*ysq  +1)
```

Thus, since y will be on the order of 1e-16, and the series is a series in the square of y, our series approximation for the log function will gain roughly 32 significant digits of precision per term. 200 significant digits comes about very quickly using such a scheme. In fact, the value of $\log(10)$ stored in HPF comes from exactly that type of computation. I first computed a 16 digit approximation, then used that to generate a 5000 digit approximation. Using that approximation to pull myself up by my bootstraps, only 10 terms in the log series was now sufficient to assure 100,000 final digits of accuracy. How many other series approximations will you find that produce 10,000 correct digits per term in your result?

```
>> L
L =
    1.09861228866810956006361266190651804208755493164062 5

L1 =
    1.09861228866810969139524523692252570464749055782256 0
68442648288985444090179615409327621073890128869190667 8
77032225675426402259013597288652146349758770138703246 1
58018597442430375013051507154995545156 54

>> L2
L2 =
    1.09861228866810969139524523692252570464749055782274 9
45173469433363749429321860896638523703971420449219385 2
43977393896208957932261979586332981221049629513625359 7
56787364577080987554032104655255311569 63

>> L3
L3 =
    1.09861228866810969139524523692252570464749055782274 9
45173469433363749429321860896687361575481373208878797 0
02906595636153612478942338639746413189149598120096511 2
65308240898440851301329051779680347997 94

>> L4
L4 =
    1.09861228866810969139524523692252570464749055782274 9
45173469433363749429321860896687361575481373208878797 0
02906595786574236800422593051982105280186572250460729 4
89135143683174153795194136247646258915 06
```

```
>> L5
L5 =
    1.0986122886681096913952452369225257046474905578227 49
45173469433363749429321860896687361575481373208878797 0
02906595786574236800422593051982105280187076727741060 3
16276918338136717935590088826893310339 92

>> L6
L6 =
    1.0986122886681096913952452369225257046474905578227 49
45173469433363749429321860896687361575481373208878797 0
02906595786574236800422593051982105280187076727741060 3
16276918338136717937369884436095990374 25
```

## 4.3 Square and Cube Roots

In the hierarchy of functions, square roots are an easy thing to compute. Of course, the method I learned long ago in school, computing them on paper in a form that looked like a long division, is one alternative.

In HPF, one might use a simple scheme based on logs, however, this requires first a log, then a divide, then an exponentiation. Note that the divide by 2 would be done as a multiply by 0.5. Since logs and exponentiations are computations that are linear in the number of digits needed, they tend to be slow.

$$\sqrt{x} = e^{\frac{\log(x)}{2}} \tag{4.14}$$

Computer computation is more simply done using the Babylonian method, also known as Heron's method.[10] Said simply, given an initial approximation, we compute a better approximation using a simple divide and average scheme, approximating the value $y = \sqrt{x}$.

$$y_{n+1} = \frac{y_n + \frac{x}{y_n}}{2} \tag{4.15}$$

A great thing here is the Babylonian method essentially doubles the number of correct digits in the result for every iteration. At the same time, it also requires a divide for every iteration, and divides are moderately expensive in HPF compared to multiplies. Can we improve things? First of all, recognize that there are range reduction tricks we can accomplish. If we represent X in a floating point format, we can reduce the exponent modulo 2, then multiplying the result by one half as many powers of 10 as we dropped off. Essentially, this means we need

[10] A derivation of this scheme comes from application of Newton's method to the root finding problem $x - y^2 = 0$. It will be quadratically convergent, thus doubling the number of correct digits per iteration.

really worry about computing the square root only of numbers in the half open interval $[1, 100)$.

The next trick is not unlike that we used for $\log(x)$. Compute a double precision approximation, then convert back to HPF form.

```
y0 = hpf(sqrt(double(x)));
```

We could use the Babylonian method, starting from that point, but it still requires divides. Is there a divide free alternative? Recall the trick we used to compute the reciprocal.

The trick comes from approximating the square root of $\frac{1}{x}$ instead. Apply Newton's method instead to $x - \frac{1}{y^2} = 0$.

$$y_{n+1} = y_n \frac{3 - xy_n^2}{2} \tag{4.16}$$

Again, the divide by 2 is really a multiply by 0.5. The iteration converges as long as a reasonable estimate is available for the desired square root. Of course, since we start with a very accurate approximation based on the double precision approximation, that is not an issue. This iteration is again quadratically convergent, doubling the number of correct digits with each pass through, but no divides are necessary. When done, compute the desired square root by one more multiply, since we know that $\sqrt{x} = x\frac{1}{\sqrt{x}} = xy$.

As an example, compute the square root of 2, reporting 200 significant digits.

```
>> y = sqrt(hpf('2',[200,4]))
y =
    1.41421356237309504880168872420969807856967187537694807317667973799073247846210703885038753432764157273501384623091229702492483605585073721264412149709993583141322266592750559275579995050115278206057147
>> y.*y
ans =
    2.
```

As evidence that the method is indeed quadratically convergent, the absolute errors after each successive iteration in the approximations for $\sqrt{2}$ were [9.6673e-17, -9.9126e-33, -1.0422e-64, -1.1521e-128, 0]. That last zero is of course not a true zero, but only zero relative to a 200 digit approximation.

Cube roots are similar, in that we can use Newton's method as applied to the function $x - \frac{1}{y^3} = 0$. The iterative scheme will again be quadratic in convergence, thus doubling the number of correct digits per iteration.

$$y_{n+1} = y_n \frac{4 - xy_n^3}{3} \tag{4.17}$$

Once this scheme has converged to the cube root of the inverse, recover the desired cube root of x as $xy^2$. Is it quadratically convergent?

```
DefaultNumberOfDigits 250 5
x = hpf('1.331');
% Compute the initial approximation
y0 = hpf(1./double(x).^(1/3))
y0 =
0.90909090909090906063028114658663980662822723388671875
00000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000

y1 = y0.*(4 - x.*y0.^3)./3
y1 =
0.90909090909090909090909090909090707393518551445849685
81023150559213349981349517897099077829394150245752338 5
33707664716774612620828792740620415909043618685408727 0
44840690738238681012362008004856761544942855834960937 5
00000000000000000000000000000000000000

y2 = y1.*(4 - x.*y1.^3)./3
y2 =
0.90909090909090909090909090909090909090909090909090909
09090909090819590866919965828143870686514312982274002 9
21446011019626543351944197362972049401017066761456452 5
39056784498296813495985461195051027332579907449554024 1
96528446835692550106346283662285992 29

y3 = y2.*(4 - x.*y2.^3)./3
y3 =
0.90909090909090909090909090909090909090909090909090909
09090909090909090909090909090909090909090909090909090909
09090909090909090909089146834248398772157754470844094 6
33270471471977071392184534537681306230606413291753929 7
44887620080544643150360277845533883 82

y4 = y3.*(4 - x.*y3.^3)./3
y4 =
0.90909090909090909090909090909090909090909090909090909
09090909090909090909090909090909090909090909090909090909
09090909090909090909090909090909090909090909090909090909
09090909090909090909090909090909090909090909090909090909
09090909090909090909090909090909091

cubrt_x = x.*y4.*y4
cubrt_x =
1.1
```

## 4.4 Sine and Cosine

For example, how might one compute the number $\sin(10^{400})$ with any given number of correct digits? In fact, this was one of the goals I put to myself when I initially wrote the HPF class, to be able to do that exact computation.

The first thing one should consider in any such problem is if a range reduction exists. It does, since the sine function is periodic, with a period of $2\pi$.

$$\sin(x) = \sin(x + 2\pi n) \tag{4.18}$$

The unavoidable cost is a divide by $2\pi$, to remove integer multiples of that number. This also costs us in terms of a massive subtractive cancellation issue. Thus one must carry more than 400 significant digits to ever compute the since of a number that large if you wish to have any significant digits remaining in the result at all. Assuming that is a given, we are left now only with the problem of how to compute $\sin(x)$, knowing that x lies in the interval $[0, 2\pi]$.

A further range reduction is available though. Essentially, we split the interval $[0, 2\pi]$ into subintervals, using a few trigonometric identities.

$$\sin(x) = \quad \cos(x - \frac{\pi}{2}) \tag{4.19}$$

$$\sin(x) = -\sin(x - \pi) \tag{4.20}$$

$$\sin(x) = -\cos(x - \frac{3\pi}{2}) \tag{4.21}$$

$$\sin(x) = \quad \sin(x - 2\pi) \tag{4.22}$$

Essentially, this allows us to reduce the problem to computation of either $\sin(x)$ or $\cos(x)$, but now the interval of interest will be reduced to $[-\frac{\pi}{4}, \frac{\pi}{4}]$ in all cases.

The trick of range reduction can be extended even further using various exact relations for the sine function. A few such values are given as ...

$$\sin(\frac{\pi}{60}) = \frac{1}{16}[2(1 - \sqrt{3})\sqrt{5 + \sqrt{5}} + \sqrt{2}(\sqrt{5} - 1)(\sqrt{3} + 1)] \tag{4.23}$$

$$\sin(\frac{\pi}{6}) = \frac{1}{2} \tag{4.24}$$

In conjunction with the sum of angles formula for $\sin(x + y)$, these exact constants can yield further significant range reductions.

$$\sin(x + y) = \sin x \cos y + \cos x \sin y \tag{4.25}$$

Regardless, at some point we must give up on the idea of range reductions once x is limited to a small enough interval. At that point, we revert to the standard Taylor series for the sine or cosine functions.

$$\sin(x) = \sum_{n=0}^{\infty} \frac{-1^n}{(2n+1)!} x^{2n+1} \quad = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots \tag{4.26}$$

$$\cos(x) = \sum_{n=0}^{\infty} \frac{-1^n}{(2n)!} x^{2n} \quad = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \cdots \tag{4.27}$$

See that these series converge rapidly for $|x| \leq \frac{\pi}{4}$, in fact more so even than the exponential series. The same trick applies as with the exponential series though. Reverse the series, factor out $n!$, then apply Horner's rule to evaluate it without recourse to a divide, except one last divide at the very end.

Finally, as a means of computing the sine function on the number I mentioned earlier, $1e400$, note that the periodicity of the sine function causes an effective loss of precision. Subtractive cancellation on a massive scale occurs here, so that the last 400 digits of the result will be floating point trash. (Yes, no matter what you do, if you try hard enough it is always possible to generate numerical trash with a computer.) However, we can force HPF to do its computational work in 1400 digits of precision, to yield a net of 1000 correct digits in the result.

```
>> sin(hpf('1e400',[1000 400]))
ans =
   -0.99853823198309772291615928069189819569651330348634
4
83348113790876119854449993176233191763176584539444995
6
40206521489685521811153252158188751164339628376139878
2
10787481169862093399686633452702153022890054781494108
6
02339797754899680730477819415085559894011235446201127
2
08600682604483223219113236020746370757495933546950140
2
66422141790099665836773140664438367521481512273712176
0
44381389302181305000751694634029879584110218548614671
2
71424910476883915598006205467978360580686102965497439
0
85919530088680629344102372344642572911716138179577722
1
32576955580433214840134091384718125165030137376083428
7
36446295562890348587354752769218298874328272818049446
6
16597373675673570415903876971467446236361281745120317
5
11569838026704485327955466932062341208957149133901093
6
71490719499825522802932670488597807222550840531305764
0
02155038110442775774927287843019311922336402818796486
8
92472840557363050347832567981801306776713941497099954
4
39588674422842381995379328030362385052140137734751918
1
06740848987123674889910633655 18
```

## 4.5 Arc Sine

Accurate and efficient computation of the inverse sine function uses another numerical trick. There is no issue of range reduction, since the domain of arcsin will be [-1,1]. (Recall that HPF is not yet defined to work in the complex domain. I'll probably leave that extension to some other individual, or perhaps my next project.)

We can start again by looking at the Taylor series.

$$\arcsin(x) = \sum_{n=0}^{\infty} \frac{(2n)!}{2^{2n}(n!)^2(2n+1)} x^{2n+1} \tag{4.28}$$

Expanded, perhaps the terms of the series are more easily visualized in (4.29). Here we see that each coefficient in the series can be generated from the previous one by multiplication of a simple fractional form. Careful inspection will also show that a divide will still be necessary at each step. This means we will want to sum as few terms as possible.

$$\arcsin(x) = x + \left(\frac{1}{2}\right)\frac{x^3}{3} + \left(\frac{1\cdot 3}{2\cdot 4}\right)\frac{x^5}{5} + \left(\frac{1\cdot 3\cdot 5}{2\cdot 4\cdot 6}\right)\frac{x^7}{7} + \cdots \tag{4.29}$$

The direct approach is to use an approximate solution, accurate to roughly 16 digits, as we did for the log function. Again, since the series terms increase by powers of $x^2$ for each term, this will gain roughly 32 digits per term in our result. The way to get that initial approximation is trivial, merely convert X to a double precision form, then let MATLAB compute the asin, then convert back in HPF form. All of these conversions are quite fast.

```
1 Y = hpf(asin(double(X)));
```

Next, employ a transformation $X = Z + Y$, using a sum of angles identity for the sin function.

$$\sin(X) = \sin(Z+Y) = \sin(Z)\cos(Y) + \cos(Z)\sin(Y) \tag{4.30}$$

If Y is known to be accurate to roughly 16 significant digits as an approximation for X, then Z must be on the order of $10^{-16}$. In (4.30), we can accurately compute $\sin(Y)$ as an HPF number. Of course, while $\cos(Y)$ would also be computable accurately, it is more efficient to compute $\cos(Y) = \sqrt{1-\sin^2(Y)}$, once $\sin(Y)$ is known.

Similarly, we must recognize that the unknown here is $\sin(Z)$, so the $\cos(Z)$ term is eliminated by use of the same identity. Finally, solve the resulting quadratic

equation for $\sin(Z)$. The right hand side of (4.31) is entirely known, and also very small, typically of the order of $10^{-16}$.

$$\sin(Z) = X\sqrt{1-\sin^2(Y)} - \sin(Y)\sqrt{1-X^2} \tag{4.31}$$

The series for inverse sine is rapidly convergent for arguments this small, yielding roughly 32 digits per term.

## 4.6 Tangent

The tangent function offers its own numerical problems to solve.The tangent function is periodic, with period $\pi$. So we must first translate any value into the proper period, here $[-\frac{\pi}{2}, \frac{\pi}{2}]$. Next, consider the series for the tangent function.

$$\tan(x) = \sum_{n=0}^{\infty} \frac{U_{2n+1}}{(2n+1)!} x^{2n+1} \tag{4.32}$$

Here $U_{2n+1}$ refers to the sequence of up/down numbers, or Euler zig-zag numbers. The exact computation of these numbers, or the Bernoulli numbers they are related to, will itself take more work than is worth expending to compute the tangent function. Therefore I'll simply go on to an alternative solution.

One could always fall back on the simple definition, as the ratio of sin(x) and cos(x).

$$\tan(x) = \frac{\sin(x)}{\cos(x)} \tag{4.33}$$

A problem is this ratio forces us to compute both trig functions, so is perhaps too costly in time. An alternative is to use a basic identity to save one of those function calls.

$$\sin^2(x) + \cos^2(x) = 1 \tag{4.34}$$

Then we can express the tangent at a cost of only one trigonometric function evaluation, plus a square root. (Note the sign of this result will need to be attended to with some care.) Square roots are efficiently computed by HPF, because of the quadratic behavior of the algorithm.

$$\tan(x) = \frac{\sqrt{1-\cos^2(x)}}{\cos(x)} \tag{4.35}$$

The flaw with (4.35) is seen when x is small. Here we will expect to see subtractive cancellation problems. For example, even in double precision arithmetic, compare these results.

```
>> format long g
>> tan(0.001)
ans =
        0.00100000033333347

>> sin(0.001)/cos(0.001)
ans =
        0.00100000033333347

>> C = cos(0.001);
>> sqrt(1-C^2)/C
ans =
         0.0010000003333145
```

See that the tangent function computes a result consistent with the ratio of the sin and cosine. However, if I use the form in (4.35) to compute the tangent, see that I've lost a few digits at the end. Why is this? It happens because the cosine function near zero is a number very near 1.

```
>> cos(0.001)
ans =
          0.999999500000042
```

Square that number (which is very near 1) and then subtract it from 1, and the beast of subtractive cancellation leaves us with a loss of significant digits. A nice trick is to compute the *versine* function instead of the cosine.

$$\text{versin}(x) = 1 - \cos(x) \tag{4.36}$$

Of course, do not compute the cosine itself, and then subtract it from 1. Instead, use the series approximation for $\cos(x)$ without the first term in the series!

$$\text{versin}(x) = \sum_{n=1}^{\infty} \frac{-1^{n+1}}{(2n)!} x^{2n} = \frac{x^2}{2!} - \frac{x^4}{4!} + \frac{x^6}{6!} - \frac{x^8}{8!} + \cdots \tag{4.37}$$

All of the tricks suggested previously for efficiently computing the sine and cosine still apply here of course. Compute that series in reverse using a Horner's rule to minimize divides. Then given $V = \text{versin}(x)$ from the above series, compute $\tan(x)$.

$$\tan(x) = \frac{\sqrt{2V - V^2}}{1 - V} \tag{4.38}$$

This scheme is both efficient and accurate.

## 4.7 Arc Tangent

Computation of the arc tangent function is not unlike many others described so far, but here too we can benefit from the sum of angles formula.

$$\tan(x) = \tan(u+v) = \frac{\tan(u)+\tan(v)}{1-\tan(u)\tan(v)} \tag{4.39}$$

Choose $v = \arctan(\texttt{double(x)})$ to be a good approximation to the desired inverse tangent of $x$. Solving for the unknown, $\tan(u)$, we arrive at a simple expression for the tangent of a quite small number.

$$\tan(u) = \frac{\tan(v)-x}{1-x\tan(v)} \tag{4.40}$$

In fact, $u$ will generally be quite small, on the order of $10^{-16}$, i.e., double precision eps. Simply compute the inverse tangent of $u$ using a series approximation, then recover $\arctan(x)$ from u and v.

$$\arctan(u) = \sum_{n=0}^{\infty} \frac{-1^n u^{2n+1}}{2n+1} = u - \frac{u^3}{3} + \frac{u^5}{5} - \frac{u^7}{7} - \cdots \tag{4.41}$$

$$\arctan(x) = u + v \tag{4.42}$$

The series (4.41) does require a divide for each term, however, since every term gains roughly 32 digits in the result, very few terms will be necessary.

## 4.8 Efficiency of series versus direct computations

Note that for some of the trigonometric functions, I have implemented them using computations based on the sin and cos functions, rather than a direct series. For example, we might compute the tangent function as simply the ratio of the sine and cosine functions (4.43), or we might use a series expansion (4.44).

$$\tan(x) = \frac{\sin(x)}{\cos(x)} \tag{4.43}$$

$$\tan(x) = \sum_{n=1}^{\infty} \frac{-1^{n-1}2^{2n}B_{2n}}{(2n)!}x^{2n-1} = x + \frac{1}{3}x^3 + \frac{2}{15}x^5 + \frac{17}{315}x^7 + \cdots \tag{4.44}$$

The sine and cosine functions are efficiently computed using series expansions, explicitly configured to avoid divides. As such, they are both efficient. In fact, for the tan function I have been even more efficient, using the versin function

to reduce that computation to a single series. How about the series expansion for the tangent function? The flaw with this series is in the computation of the Bernoulli numbers, a sequence that will itself require some effort to generate if there are many terms necessary in the expansion. If the coefficients of the series will themselves be HPF numbers, then the series will be expensive to compute for a large number of terms. This is a tradeoff that must be taken to find the most efficient scheme for computation, yielding the desired accuracy.

# 5 Missing?

There are infinitely many special functions I might have included in the HPF tools. At some point, one must give up, be willing to stop writing code at least temporarily. Some special functions that I know to be missing are things like the gamma function, beta function, erfc, and erfinv. Furthermore, I know that my computation of erf is not as fast as I would like it to be. So there are many places for enhancements in HPF, but such a tool is necessarily open ended. Please accept my apologies if I have not included your favorite special function in HPF, but note that HPF is extensible.