Numerical Analysis **Irvin M. Miller, Ph.D. © 2014**

Introduction

We soon discover that it take much time to do calculations that require many digits to perform the calculation to a high degree of accuracy. Without the knowledge of upper math, it is tedious to do these calculations. With calculus we can learn many shortcuts and appreciate using these shortcuts to get the answer. We also learn that many problems cannot be solved directly with algebra, but that there are numerical techniques that will permit us to do get an answer. Even with the use of a calculator, we find that our frustration remains. However, programming a computer becomes an ideal way to further enhance our understanding of mathematics. With programming, we must describe step by step what the computer must do. This process can only be done if we understand how to do it without the computer. We are going to illustrate this reasoning by studying how to generate logarithmic and trigonometric tables.

We would like to find $10^{.333333}$. we could do this by the technique of using the binary number system and taking square roots of 10:

1	1.	10	.333333
1/2	.5	3.162277	-.250000
1/4	.25	1.778279*	.083333
1/8	.125	1.3335214	-.062533
1/16	.0625	1.1547819*	.020833
1/32	.03125	1.07460782	-.015625
1/64	.015625	1.03663292*	.005108
1/128	.0079125	1.01815172	-.003956
1/256	.00395625	1.00903504*	.001152

$10^{.333333}$=1.7782*1.1546*1.0366*1.0090=2.1474

From calculus we know $(1+a)^n=1+na+n(n-1)a^2/2!+n(n-1)(n-2)a^3/3!+..$
Let a = 2/8 and n=1/3. then:
$(1+2/8)^{1/3}=(10)^{1/3}/2=(1+1/4)^{1/3}$=1+.25/3-.0625/9+5*.015625/81=1.08333-.00694+.00095
=1.07639+.00095=1.07734

Therefore, $10^{1/3}$=2*1.07658=2.15468 (2.1544)
As we can see the non-advanced approach takes more calculations of greater complexity than the advanced approach.

Square roots

Let us begin our lessons by learning how to take a square root.

$$x^2=b$$

Adding x^2 to both sides of the equation:

$$2x^2=x^2+b$$

Then, $x=(x^2+b)/2x$

The above equation is a recursive equation, where we use a value of x to calculate a new value of x and then that one to calculate another new one.

Start with x=1:

$x=(1^2+2)/(2*1)=1.5$

$x=(1.5^2+2)/(2*1.5)=4.25/3=1.416..$

$x=(1.416^2+2)/(2*1.416)=4.005056/2.832=1.414214$ (1.414213..)

We see that in 3 steps, we have the answer accurate to 5 decimal places. In the next step, it would be ten places. While these calculations are manageable fro the first few steps, they become more difficult with greater iterations. Let us write a computer program in python:

```
x=1.
k=0
while k<3:
   x=(x*x+2)/(2.*x)
   k=k+1
print x
```

This six line program is the instructions that describe what we just did. We assign the value of 1. to x and the value of 0 to k. The while statement tells the computer to continue executing the instruction up to but not including the print statement. Once k reaches of value greater than 3, the loop is terminated, and we go onto the print instruction. The **colon** tells us that the **while** statement is a branch instruction, and the indentation tells us these intstructions are part of the while loop. At the end of these three iterations, x=1.41421568627 Let us change the program as follows:

```
x=1.
k=0
while k<7:
   x=(x*x+2)/(2.*x)
   k=k+1
   print x
```

The output will be:

```
1.5
1.41666666667
1.41421568627
1.41421356237
1.41421356237
1.41421356237
1.41421356237
```

We see that on the fifth iteration, the computer calculated the answer to 12 places.
We are learning to program by example.

Subroutines

We would like to use this square root routine over and over without having to write the code over and over. We achieve this goal by making a subroutine

```
def mysqrt(b):
   x=1.
   k=0
   while k<3:
      x=.5*(x+b/x)     # this line of code was replaced with its equivalent,
      k=k+1
   return x
```

Because the subroutine is like a branch, it has to end in a colon, and the next lines of code have to be indented 4 spaces. The # denotes a comment or statement that is not to be executed. The b is the number of which we which to find the square root. Let us now run a test program.

```
def mysqrt(b):
   x=1.
   k=0
   while k<3:
      x=.5*(x+b/x)
      k=k+1
   return x

k=1
while k<101:
   x=mysqrt(k)
   x2=x*x
   err=x2-k
   print k,x,x2,err
   k=k+1
```

You will notice that the error gets bigger as the numbers get bigger. To make this more general the code is written as follows:

```
def mysqrt(b):
    num=b
    if num= =0.:
        return num    # sq root zero = zero
    if num<0.:        # make number positive
        num=-num
    pt=0
    while num<100.:  # bring large number in 1 to 100 range
         pt=pt+1
         num>num/100.
```

```
    while num<1.:     # bring small number in 1 to 100 range
        pt=pt-1
        num=num*100
    x=(1.+num)/2.     # for number close to 1
    if x>9.:          # for number greater than 17
        x=9
    k=0
    while k<7:
        x=.5*(x+b/x)
        k=k+1
    while pt<0:       # restoring exponent for small nos.
        x=x/10
        pt=pt+1
    while pt>0:        # restoring exponent for large nos.
         x=x*10
         pt=pt-1
    return x           # return answer
```

You can see that the code has quadrupled in size. We introduced another branch instruction: the **if** statement. Coding is an interactive process. As we execute the programs we have to learn how to deal with the anomalies and errors. This process greatly increases our feel and understanding for numbers. We can see how we used algebra to simplify expressions and understand how precise we must be when giving instructions to a computer. We by no means have learned the full language of python, but we have learned enough to write some very sophisticated programs.

The original problem

To determine the value of a number to a decimal power, we convert the exponent to powers of two to determine what roots of that number, we must multiply together. The basic code is as follows:

```
def mypow(b,exi):
    ex=exi
    p2=1.
    pt=b
    ans=1.
    k=0
    while k<53
        pt=mysqrt(pt)    # find sq. rt. of next no. in table
        p2=p2/2.          # find next power of 1/2
        if ex>=p2:        # next 2 steps are the key steps
              ex=ex-pt    # subtract power of two
              ans=ans*pt # multiply power of ten
        k=k+1
     return ans
```

As you can see while it takes two steps to define the process, we need 14 lines of code. Fortunately, we did not have to insert the code for the square root. This code only works for exponents that are positive and less than 1. We are not going to expound further because we are going to find a more efficient way of doing this using higher mathematics. Here is a sample use of the code:

Include code here for mysqrt.

```
def mypow(b,exi):
    ex=exi
    p2=1.
    pt=b
    ans=1.
    k=0
    while k<53
        pt=mysqrt(pt)   # find sq. rt. of next no. in table
        p2=p2/2.         # find next power of 1/2
        if ex>=p2:       # next 2 steps are the key steps
              ex=ex-pt   # subtract power of two
              ans=ans*pt # multiply power of ten
        k=k+1
     return ans

print mypow(10.,.375)
print mypow(10.,.30103)
```

You can begin experimenting with the code. If you put a print statement after ans=ans*pt, you can see how pt approaches 1 and p1 approaches zero. This concept begins to give you and introduction to calculus. If we reverse the process, with a small modification we can use this code to find the logarithm of a number:

```
def myln(b,num):
    ex=num
    p2=1.
    pt=b
    ans=0.            # one of the 4 lines of code changed
    k=0
    while k<53:
        pt=mysqrt(pt)
        p2=p2/2.
        if ex>=pt:       # next 2 steps are the key steps
              ex=ex/pt   # divide by power of ten
              ans=ans+p2 # add power of two
        k=k+1
     return ans

print myln(10.,2.)
```

Once we have the reciprocal function, we can then use that code to check both itself and the original function:

```
k=1
while k<11:
    x=myln(10.,k)
    y=mypow(10.,x)
    err=y-k
    print k,x,y,err
```

Exponents (scientific notation) and logarithms

Let us begin by finding a number raised to a power. In calculus, we learn that you can write the series for ex as follows: $e^x=x^0/0!+x^1/1!+x^2/2!+x^3/3!+$
For the computer, we would like to write this as a recursive function:

```
c=1.
k=11
while k>0:
    c=1+c*x/k
    k=k-1
```

Let us check this out starting with k=3

k=3: $c=1+1*x/3$

k=2: $c=1+(1+x/3)*x/2=1+x/2+x^2/(3*2)$

k=1: $c=1+(1+x/2+x^2/(2*3))*x/1=1+x/1+x^2/(2*1)+x^3/(3*2*1)=1+x/1+x^2/2!+x^3/3!$

If we want the series to converge rapidly so that we don not have to use lots of terms, then we want to use small values for x. Thus if I have e^{n+x} where x is a number less than 1. but greater than zero and n is an integer, we have e^n*e^x. But, e^x can be written as: $(e^{x-a+a}=e^{x-a}e^a)$. However, we need to know the value of e. We can use our recursive function, where x =1, and we start k=30.

Let us check out the theory with numbers:
$e^{.69314}=e^{11*.0625+.00564}=(e^{.0625})^{11}*e^{.00564}=(1.06449)^{11}*(1+.00564+.00564^2/2+\ldots)$
$=1.98864*1.00565=1.99987$ note: that $a^{11}=a^8xa^2xa^1$ if $a=e^{1/16}$, then $a11=e^{1/2}*e^{1/8}*e^{1/16}$
This evaluation requires 4 square roots and 4 multiplications to get the answer accurate to 4 or 5 places. If we used the series directly , we find that $.69314^{20}=.0006$. It would take over twenty terms in the series to achieve the same results as we did with two terms in the series. Remember that it took thirty terms to evaluate e.

```
def mye():
    e=.1
    x=30.
    while x>0.:
        e=1.+(1./x)*e
        x=x-1.
    return e
```

```
def myexp(xi):
    e=mye()            # find e=2.718281828459045
    x=xi
    exs=1
    if x<0.:           # treat as positive exp, invert ans
        x=-x
        exs=-1
    ex=int(x)          # find integer portion of exponent
    x=x-float(ex)      # find decimal portion
    e16=e
    i=0                # find 16th root of e
    while i<4:
        e8=mysqrt(e16)
        i=i+1
    sixtth=1./16.      # make x less than 1/16
    num=1.
    while x>=sixtth:
        x=x-sixtth
        num=num*e8
    c=1.               # find e^x
    k=11.
    while k>0:
        c=1.+c*x/k
        k=k-1.
    while ex>0:        # raising e to the integer power
        c=c*e
        ex=ex-1
    ans=c*num      # integer part times fractional part
    if exs==-1:    # if exponent was negative, invert answer
        ans=1./ans
    return ans
```

We see that we need to know the mathematical theory in order to program our functions so that we get reasonably accurate answers quickly.

Now we are ready to write a program to find the logarithm of a number. The logarithm is the inverse process for e^x.

From calculus we know that $\ln(1+x)=x-x^2/2+x3/3+$ Then $\ln(1-x)=-(x+x^2/2+x^3/3+\ )$
$\ln((1-x)/(1+x))=\ln(1-x)-\ln(1+x)=-2*(x+x^3/3+x^5/5+\ )$

If a= (1-x)/(1+x) x=(1-a)/(1+a)
Then ln(a)= -2*(x+x^3/3+x^5/5+)=-2x*(1+x^2/3+x^4/5+)

```
def myln(xi):
    e=mye()        # find value of e
    inv=0
    x=xi
    if x<1.:       # if xi is less than 1, invert
       inv=1
       x=1./x
    root=e
    i=0
    while i<5:     # Find  1/32 root of e
        root=mysqrt(root)
        i=i+1
    i=0.           # make operand less than to find integer
    while x>=e:    # portion of answer
        x=x/e
        i=i+1.
    j=0.           # make operand less than e^1/32=1.0317
    while x>=root:
        x=x/root
        j=j+1.
    x=man(x)
    x=x+i+j/32.
    if inv==1:
       x=-x
    return x

def man(xi):
    x=xi
    x=(x-1.)/(x+1.)
    x2=x*x
    k=11.
    c=x2/k
    k=k-2.
    while k>1.:
        c=(1./k+c)*x2
        k=k-2.
    c=2*(1.+c)*x
    return c
```

Let us use this program to evaluate the ln(10).
We divided by e twice to get a value less than e. 10/(e*e)=1.35335
Now we divide 1.35335 by $e^{1/32}$ until we get a number less than $e^{1/32}$.
This takes 9 divisions giving us: 1.02156 So far we have 2+9/32=2.28125
The x=(1.02156-1)/(1+1.02156)=.010665 Using the power series, we get:

$2*.010665*(1+.010665^2/3+.010665^4/5+ =.02133*(1+.00003+)=.02133$
The final answer is 2.28125+.02133= 2.30258 This numerical check is important because it helps us understand the process better and to check that the theory truly indeed works.

These two functions become the foundation for a more general problem such as:
$y=10^{.47712}$
We will leave it to you to write the program mygenpow(b,ex):
$\ln(y)=.47712*\ln(10)= .47712*2.30258=1.098606$
$y=e^{\ln(y)}=e^{1.098606} =2.99998$ using mypow
Also $e^{1.098606}=e*e^{.098606}=2.71828*1.10363=2.99997$

Reflections:

In school, we learn all of the pieces of knowledge that we need in order to do what has just been described. But, until we actually do the calculations, we only have a tangential knowledge of the topic rather than an in depth knowledge. The checking of the program can only be done with hand calculations to give as a feel for the limits of accuracy. However, in order to accomplish the task, we really had to understand the theory of exponents and logarithms. The calculations and programming help us understand why the theory and derivations were so important. As a computer professional, I was able to feel errors in computer hardware and software. There are better ways of programming these functions. With the knowledge that you have obtained from these lessons, you should be able to compare and analyze different approaches.

Trigonometry

The part of trigonometry in which we prove the equivalence of different relationships has been a foundation for integral calculus and differential equations. However, these formulae are also important for developing programs to evaluation trigonometric functions. First we begin by finding the value of pi. The formula for the half angle is a s follows: $\cos(\theta/2)=(1+\cos(\theta))/2)^{1/2}$ and $\sin(\theta/2)=((1-\cos(\theta))/2)^{1/2}$ we know that $\cos(45^0)=\sin(45^0)=2^{1/2}/2$ Using these formula, we find that:

$\tan(22.5^0)=((2-2^{1/2})/(2+2^{1/2}))^{1/2}$

$\tan(11.25^0)=((2-(2+2^{1/2})^{1/2})/(2+(2+2^{1/2})^{1/2}))^{1/2}$ $11.5^0=pi/16$

We can know see a pattern. The series for evaluating the arctan is:

$\text{atan}(x)=x*(1-x^2/3+x^4/5-x^6/7+$

Therefore if $c=((2-(2+2^{1/2})^{1/2})/(2+(2+2^{1/2})^{1/2}))^{1/2}$ $pi=16*\text{atan}(c)$
You will notice from the code we had to go the tenth ((21-1)/2) term in the series.

```
def mypi():
    i=2
    c=mysqrt(2.)
    while i>0:
        x=2.-c
        s=mysqrt(x)
        x=2.+c
        c=mysqrt(x)
        i=i-1
    x=s/c
    x2=x*x
    k=21.
    hold=1./k
    k=k-2.
    while k>0:
        hold=1./k-x2*hold
        k=k-2.
    hold=(x*hold)*16.
    return hold
```

The power series for the sine and cosine are:

$\sin(x)=x*(1-x^2/3!+x^4/5!-x^6/7!+\ )$

$\cos(x)=1-x^2/2!+x^4/4!-x^6/6!+\ )$

Remember that $\cos(90^0-x)=\sin(x)$

The code for the cosine in degrees is:

```
def mycos(i):
    pie=mypi()
    if(i>45.):
        x=(90.-i)*pie/180.
        x2=x*x
        c=1.
        j=16.
        while j>0.:
            c=1.-(x2/(j*(j+1.)))*c
            j=j-2.
        c=x*c
    else:
        x=i*pie/180.
        x2=x*x
        c=1.
        j=14.
        while j>0.:
            c=1.-(x2/(j*(j-1.)))*c
            j=j-2.
    return c
```

To complete the section, we need to find the arccosine of a variable. The code for this is:

```
def myacos(xi):
    x=xi
    pie=mypi()
    sw=1              # angle greater than 55 degrees
    if x>.57:
        sw=2          # angle btwn 35 and 55 degrees
    if x>.81:
        sw=3          # angle less than 35 degrees
        x=(x+1.)/2.
        x=mysqrt(x)   # cos of half angle
        c=(1.+x)*(1.-x)
        x=mysqrt(c)   # sine of half angle
    if sw==2:
        x=1.-2.*x*x   # y=cos(45+x) sin(2x)=1-2y*y
    if sw==1:
        x=(1.+x)*(1.-x)
        x=mysqrt(x)   # sine of angle
        x=(1.-x)/2.
        x=mysqrt(x)   # sine of half angle
    x2=x*x
    k=27.
    c=1./k
    k=k-2.            # arcsine series
    while k>0.:
        c=(1./k)+((k*x2*c)/(k+1))
        k=k-2.
    c=((x*c)*180.)/pie
    if sw==3:
        c=2.*c        # get full angle
    if sw==2:
        c=45+c/2.     # back to original angle
    if sw==1:
        c=90.-2.*c    # get full angle
    return c
```

There was a slight problem with convergence near 45^0. We thus made use of the formula $y=\cos(45+x)=\cos(45)\cos(x)-\sin(45)\sin(x) = (\cos(x)-\sin(x))/2^{1/2}$

$2y^2=\cos^2(x)-2\sin(x)\cos(x)+\sin^2(x)= 1-\sin(2x)$

$\sin(2x)=1-2y^2$

Many problems that we see in the textbook are generated as a result of a real problem that an applied mathematician solved.

Floating point arithmetic

Many modern computers use a word length of 64 bits to store a floating point number. A floating point number is the implementation of the scientific notation with the exponent being placed before the mantissa rather than after it, For calculations a 110 bit word is needed. Since a computer uses a base 2 arithmetic, numbers must be converted back and forth between base 10. The leading 8 bits or so is used for the exponent with the first bit being the sign of the number and the second bit in theory being the sign of the exponent. This would leave us with 6 bits for the exponent. Since six bits allows a number up to 63, the number is $2^{64}=(2^{10})^{6.4}=(10^{3})^{6.4}= 10^{19.2}$ With 6 bits we get a max number of 10^{19}, with 7, 10^{38} and with 8, 10^{76}. With the mantissa we have 56 bits. Since $2^{56}=(2^{10})5.6 =(10^{3})^{5.6}=10^{16.8}$, then we get a maximum of 16 to 17 decimal digits. Note that $64*\log_{10}(2)=19.2$ and that $56*\log_{10}(2)=16.8$ There are several schemes for implementing floating point numbers. I will just so you a few simple ones where the decimal point separates the exponent from the mantissa.

01000001.100000_2 =2^(1000001_2-1000000_2)x 1 /2 =2x1/2=1

01000100.101000_2==2^4x(1/2+1/8)=16*(1/2+1/8)=10

11000100.101000_2=-10

01000000.1100002=2^0*(1/2+1/4)=.75

Summary

As we can see from the examples, numerical analysis builds upon the theory of mathematics. The standard series for the logarithm does not converge for numbers greater than 1, making it in its raw form impractical. We are required to use our imaginations to transform this formula so that it can be used. When we perform lots of calculations, the round off error can seriously degrade the accuracy of our answer. When we perform lots of calculations, speed becomes important. Computers use floating point arithmetic with a fixed length so that they can get speed, but this limits the accuracy which for most engineer problems, does not become a hindrance.

In the course of learning how to derive numerical answers, we began to understand the topics of logarithms, exponents and trigonometric functions better.