

Chapter 5

Computing with Multiple Precision

In this section we shall show how to perform some computations with more digits than given by the IEEE floating point arithmetic. The problems of this section will need integer operations and variables of integer data types. It is an opportunity to learn the corresponding MATLAB features.

5.1 Computation of the Euler Number e

We shall compute the Euler number $e = \exp(1)$ to an arbitrary number of decimal digits. For this we will use the algorithm e1 in Chap. 4 which we developed to compute the exponential function using the Taylor series. The series is evaluated for $x = 1$:

$$e = \sum_{k=0}^{\infty} \frac{1}{k!}. \quad (5.1)$$

Using the notations $a = 1/k!$ for the k th term and s for a partial sum we get the function

```
function s=Eulerconstant;
s=1;sn=2; a=1; k=1;    % initialization
while s~=sn
    s=sn; k=k+1;
    a=a/k;              % new term
    sn=s+a;             % new partial sum
end
```

Indeed we obtain with

```
>> format long
>> s=Eulerconstant
s =
    2.718281828459046
```

a result with 16 decimal digits which is what we can expect by using IEEE-arithmetic. If we want to compute more digits we need to simulate multiple precision arithmetic. In the above algorithm the only arithmetic operations are

$$k = k + 1, \quad a = a/k \quad \text{and} \quad sn = s + a.$$

For k we may use a simple variable. We shall not compute so many terms of the Taylor series that also k has to be represented in multiple precision arithmetic. The partial sum and the terms to be added, however, have to be *multiple precision numbers*. We shall store the digits of a multiple precision number in a integer array. There are several integer data types in MATLAB available. Here we shall use `uint32`¹, which is a data type for unsigned integers.

This function is used for conversion to 32-bit unsigned integers. These 32-bit numbers cover the range from 0 to $2^{32} - 1 = 4294967295$. Let a be an array of such unsigned integer numbers. We represent the number 1 using 20 digits:

```
>> a=uint32(zeros(1,20));
>> a(1)=1
a =
Columns 1 through 10
    1     0     0     0     0     0     0     0     0     0
Columns 11 through 20
    0     0     0     0     0     0     0     0     0     0
```

If we wish to divide this number by $k=2$ we should get the result

```
a =
Columns 1 through 10
    0     5     0     0     0     0     0     0     0     0
Columns 11 through 20
    0     0     0     0     0     0     0     0     0     0
```

Another division by $k=3$ should give

```
a =
Columns 1 through 10
    0     1     6     6     6     6     6     6     6     6
Columns 11 through 20
    6     6     6     6     6     6     6     6     6     6
```

¹For Numeric MATLAB Types see <http://www.mathworks.com/help/matlab/numeric-types.html>

We would like to see the digits not as elements of a vector but continuously as large number. This can be done by using the function `sprintf` (`sprintf` formats data into a string of characters).

```
>> sprintf('%01d',a)
ans =
016666666666666666666666666666666666
```

Let us program this division. We need to use a function for integer division. In MATLAB this is the function `idivide`. For the remainder we use the function `mod`. Suppose we want to divide 14 by 3. The result is: $\text{quotient} = 14/3 = 4$ and $\text{remainder} = \text{mod}(14,3) = 2$. Programmed in MATLAB this is

```
quotient=idivide(14,3)  remainder=mod(14,3)
```

Thus our division function becomes

```
function a=Divide1(k,a)
% divides the integer array a by integer number k
n=length(a);
c=10;
remainder=a(1);
for i=1:n-1
    a(i)=idivide(remainder,k);
    remainder=mod(remainder,k)*c+a(i+1);
end
a(n)=idivide(remainder,k);
```

Let us test this function. The following script divides the initial number $a=1$ with the numbers $k = 2, \dots, 15$. Thus the last result should be $1/15!$:

```
clear, clc, format long
res=[];
a=zeros(1,30,'uint32');
a(1)=1;
for k=2:15
    a=Divide1(k,a);
    res=[res;sprintf('%01d',a)];
end
res
1/factorial(15)

res =
050000000000000000000000000000000000
016666666666666666666666666666666666
004166666666666666666666666666666666
000833333333333333333333333333333333
000138888888888888888888888888888888
000019841269841269841269841269841269
000002480158730158730158730158730158
```

```

0000000275573192239858906525573
0000000027557319223985890652557
000000002505210838544171877505
000000000208767569878680989792
000000000016059043836821614599
000000000001147074559772972471
000000000000076471637318198164
>> 1/factorial(15)
ans =
    7.647163731819816e-13

```

It looks good! Notice that the smaller the numbers get the more leading zeros appear. It is not necessary to divide these leading zeros by k since they remain zero. We use the variable `imin` to count the number of leading zeros in the vector and start the division at position `a(imin+1)`. The division function changes so to

```

function [A,imin]=Divide(c,imin,k,A)
% DIVISION divides the multiple precision number A by the integer
% number k. The first imin components of A are zero. imin is updated
% after the division. c defines the number of decimal digits in one
% array element: c=10 is used for one digit, c=100 for two digits etc.
n=length(A);
if imin <n                                % if imin=n => A=0
    first=1;
    remainder=A(imin+1);
    for i=imin+1:n
        A(i)=idivide(remainder,k);
        if A(i)==0
            if first                                % update imin
                imin=i;
            end
        else
            first=0;
        end
        if i<n
            remainder=mod(remainder,k)*c+A(i+1);
        end
    end
end
end

```

Notice at the beginning of the division the variable `imin` is updated if `a(i)` becomes zero. After the first nonzero element the update is stopped. The following test shows that `imin` counts the leading zeros correctly:

```

% Testprogram for Divide for c=10 or c=100
clear, clc, format long
a=zeros(1,30,'uint32');
imin=0;
a(1)=1;
c=100;
if c==10, w='%01d'; else w='%02d'; end

```


Our main program becomes

```
function s=EmultPrec(c,n);
% EMULTPREC computes n*log10(c) decimal digits of
% the Euler constant e=exp(1). The digits are stored
% in the array s.
a=zeros(1,n,'uint32');           % define array of
s=a;                             % unsigned integers
a(1)=1; s(1)=2;
k=1; imin=0;                     % imin skips leading zeros
while imin<n
    k=k+1;
    [a,imin]=Divide(c,imin,k,a); % new term
    s=Add(imin,s,a);             % new partial sum
    s=Carry(c,s);
end
```

With these preparations we can now compute

```
>> s=EmultPrec(10,10)
s =
    2    7    1    8    2    8    1    8    2    3
>> e=sprintf('%01d',s)
e =
2718281823
```

To compute more digits we use

```
>> s=EmultPrec(10,60); e=sprintf('%01d',s)
e =
271828182845904523536028747135266249775724709369995957496673
```

Because of Rounding errors some of the last printed digits are not correct. We can check this by computing 10 more digits:

```
>> s=EmultPrec(10,70); e=sprintf('%01d',s)
e =
2718281828459045235360287471352662497757247093699959574966967627724050
```

So we see that the last two digits of `s=EmultPrec(10,60)` are affected by rounding errors.

Packing More Digits in One Array Element

The parameter `c` of `EmultPrec` controls how many digits are stored in one array element. If we change it to `c=100` we work with two digits per array element and get

```
>> s=EmultPrec(100,30); e=sprintf('%01d',s)
e =
27182818284594523536287471352662497757247936999595749646
```

Note that the printed result is wrong! Zeros are missing, e.g. for the sequence after the 13th digit we get 5945 instead of 59045. We have to adjust the format to print 2 digits with leading zero if necessary:

```
>> s=EmultPrec(100,30); e=sprintf('%02d',s)
e =
027182818284590452353602874713526624977572470936999595749646
```

Now the result is correct.

5.2 MATLAB-Elements Used in This Chapter

- uint32:** Convert to 32-bit unsigned integer.
`intArray = uint32(array)` converts the elements of an array into unsigned 32-bit (4-byte) integers of class `uint32`.
`intArray`: Array of class `uint32`. Values range from 0 to $2^{32} - 1$
- zeros:** Create array of all zeros.
`X = zeros(sz)` returns an array of zeros where size vector `sz` defines `size(X)`.
For example, `zeros([2 3])` returns a 2-by-3 matrix.
`X = zeros(1,3,'uint32')`
Create a 1-by-3 vector of zeros whose elements are 32-bit unsigned integers.
- idivide:** Integer division with rounding option.
`C = idivide(A, B)` is the same as `A./B` except that fractional quotients are rounded toward zero to the nearest integers.
- mod:** Modulus after division.
`M = mod(X,Y)` returns the modulus after division of `X` by `Y`. In general, if `Y` does not equal 0, `M = mod(X,Y)` returns `X - n.*Y`, where `n = floor(X./Y)`.
- sprintf:** Format data into string
`str = sprintf(formatSpec,A1,...,An)` formats the data in arrays `A1,...,An` according to `formatSpec` in column order, and returns the results to string `str`.

5.3 Problems

For the following problems, make use of the functions we developed for computing Euler's number e .

1. Compute using multiple precision the powers of 2:

$$2^i, \quad i = 1, 2, \dots, 300.$$

2. Write a program to compute factorials using multiple precision:

$$n!, \quad n = 1, 2, \dots, 200.$$

3. Compute π to 1000 decimal digits. Use the relation by C. Størmer:

$$\pi = 24 \arctan \frac{1}{8} + 8 \arctan \frac{1}{57} + 4 \arctan \frac{1}{239}.$$

Hints:

- Compute first a multiprecision arctan function using the Taylor-series (4.8) as proposed in Chap. 4:

$$\arctan x = \sum_{k=0}^{\infty} (-1)^k \frac{x^{2k+1}}{2k+1} = x - \frac{x^3}{3} + \frac{x^5}{5} - \dots$$

- The above series is alternating so there is a danger of cancellation. However, since it is used only for $|x| < 1$ this is not much a concern. What we need is a new function `Sub`

```
function r=Sub(c,a,b)
% SUB computes r=a-b where a and b are multiprecision numbers
% with a>b.
```

to subtract two multiprecision numbers. One has to be careful not to generate negative numbers, all intermediate results have to remain positive.

- To compute π we have to evaluate for some integer $p > 1$ the function $\arctan(1/p)$. When generating the next term after

$$t_k = \frac{x^{2k+1}}{2k+1}$$

for $x = 1/p$ we have to form

$$t_{k+1} = t_k/p^2/(2k+1).$$

There is bug that one has to avoid: by dividing the last term twice by p and a third time by $2k+1$ the variable `imin` is updated. For the next term we need to know the value of `imin` before the division by $2k+1$! Otherwise we will get erroneous results when forming t_k/p^2 .

Learning MATLAB

A Problem Solving Approach

Gander, W.

2015, XIV, 149 p. 49 illus., 7 illus. in color., Softcover

ISBN: 978-3-319-25326-8