Floating-point arithmetic

Academic year 2020-2021

System

```
> restart:
  Digits := 10:
```

1.7. Conversion between binary representation and floating-point

Exercise: Convert the 32-bit word "11000100111000000001100000000000" into its single-precision floating-point counterpart. Subsequently, convert the number 10/3 to its single-precision binary representation. Can you represent this number exactly?

Single-precision

For the binary representation of a single-precision floating-point number a 32-bit word is used.

- bit 1: Sign of the number.
- bits 2-9: Exponent of the number (don't forget the bias!).
- bits 10-32: Mantissa or fractional part of the number.

A normalized floating-point is then expressed as

$$(-1)^s * (1+f) * 2^e$$

where f is the mantissa, e the exponent and 2 the basis (since we work in binary).

For double precision a 64-bit word is used.

- bit 1: Sign of the number.
- bits 2-12: Exponent of the number (don't forget the bias!).
- bits 13-64: Mantissa or fractional part of the number.

Binary to floating-point

```
> float_bin := "11000100111000000001100000000000";
```

$$float\_bin := \text{"11000100111000000001100000000000"} \tag{2.2.1}$$

```
> sign_bin := parse(float_bin[1]);
  exp_bin := parse(float_bin[2..9]);
  mant_bin := parse(float_bin[10..32]);
```

$$sign\_bin := 1$$

$$exp\_bin := 10001001$$

$$mant\_bin := 11000000001100000000000 \tag{2.2.2}$$

```
> sign_float := (-1)^sign_bin;
```

$$sign\_float := -1 \tag{2.2.3}$$

```
> bias := 127;
  exp_float := convert(exp_bin,decimal,2);
  exp_float := exp_float - bias;
```

$$bias := 127$$

$$exp\_float := 137$$

$$exp\_float := 10 \tag{2.2.4}$$

```
> mant_float := convert(mant_bin,decimal,2)*2^(-23);
  mant_float := add((1/2)^i*parse(float_bin[9+i]),i=1..23);
```

$$mant\_float := \frac{3075}{4096}$$

$$mant\_float := \frac{3075}{4096} \tag{2.2.5}$$

```
> float_number := sign_float*(1+mant_float)*2^(exp_float);
  float_number := evalf(float_number);
```

$$float\_number := -\frac{7171}{4}$$

$$float\_number := -1792.750000 \tag{2.2.6}$$

Real number to floating-point

```
> number := 10/3;
```

$$number := \frac{10}{3} \tag{2.3.1}$$

The sign is positive -- The bit is 0.

```
> sign_bin := 0;
```

(2.3.2)

$$sign\_bin := 0 \qquad \textbf{(2.3.2)}$$

We have 2 < 10/3 < 4 -- Hence the exponent has to be 1. However, we can't forget to add the bias to really obtain the stored 8-bit word.

```
> exp_number := 1;
  exp_number := exp_number + bias;
  exp_bin := convert(exp_number,binary);
```

$$exp\_number := 1$$

$$exp\_number := 128$$

$$exp\_bin := 10000000 \qquad \textbf{(2.3.3)}$$

If we would work exactly, we would find that

(1+f) * 2 = 10/3 => f = 2/3 = 1/2 + 1/8 + 1/32 + 1/128 + ...

However, we have to truncate this sequence somewhere since we only have a finite number of bits. Should we afterwards round up or not?

```
> mant_float1 := add(1/2*(1/4)^i,i=0..11);
  mant_bin1 := convert(mant_float1*2^23,binary);
  mant_bin2 := convert(mant_float1*2^23+1,binary);
```

$$mant\_float1 := \frac{5592405}{8388608}$$

$$mant\_bin1 := 10101010101010101010101$$

$$mant\_bin2 := 10101010101010101010110 \qquad \textbf{(2.3.4)}$$

```
> float_bin1 := cat(convert(sign_bin,string),convert(exp_bin,
  string),convert(mant_bin1,string));
  float_bin2 := cat(convert(sign_bin,string),convert(exp_bin,
  string),convert(mant_bin2,string));
```

$$float\_bin1 := "01000000010101010101010101010101"$$

$$float\_bin2 := "01000000010101010101010101010110" \qquad \textbf{(2.3.5)}$$

```
> convert_bin_float := proc(float_bin)

     description "convert floating number in binary
  representation to decimal representation";
     local float_bin_str, sign_bin, exp_bin, mant_bin,
  sign_float, bias, exp_float, mant_float, float_number;

     sign_bin := parse(float_bin[1]);
     exp_bin := parse(float_bin[2..9]);
     mant_bin := parse(float_bin[10..32]);

     sign_float := (-1)^sign_bin;
     bias := 127;
     exp_float := convert(exp_bin,decimal,2);
     exp_float := exp_float - bias;
     mant_float := convert(mant_bin,decimal,2)*2^(-23);
     mant_float := add((1/2)^i*parse(float_bin[9+i]),i=1..23);

     float_number := sign_float*(1+mant_float)*2^(exp_float);
     float_number := evalf(float_number);
```

```
end proc:
```

```
> float_d10 := evalf(10/3);
  float1 := convert_bin_float(float_bin1);
  float2 := convert_bin_float(float_bin2);
```

$$float\_d10 := 3.333333333$$

$$float1 := 3.333333254$$

$$float2 := 3.333333492 \qquad \textbf{(2.3.6)}$$

If $\tilde{x}$ is the computed solution and x^* is the exact solution, then the absolute error is given by $e = x^* - \tilde{x}$ and the relative error is given by $\frac{\|x^* - \tilde{x}\|}{\|x^*\|}$.

In most cases only the relative error is meaningful:

A = 0.01, A' = 0.015 => absolute error = 0.005, relative error = 0.5 = 50%

```
> abs(float1-float_d10)/abs(float_d10);
  abs(float2-float_d10)/abs(float_d10);
```

$$2.370000000\ 10^{-8}$$

$$4.770000000\ 10^{-8} \qquad \textbf{(2.3.7)}$$

1.7. Basis and machine epsilon

Exercise: What is the basis of the hardware floating-point arithmetic in Maple? What is its value for the machine epsilon?

What is the basis for regular floating-point arithmetic in Maple? What is its value for the machine epsilon? Do you find the same results as before?

Hardware floats

To find the number base of floating-point computations we can look at its behaviour. We will use Malcolm's algortihm to determine the base.

```
> one := evalhf(1):
  zero := evalhf(0):
  A := one:
  B := one:
  while evalhf(B-one) = zero do
                        A := evalhf(A + A);
                        B := evalhf(A + one);
                        B := evalhf(B - A);
  end do:
  base := 0:
  C := one:
  while base = 0 do
```

```
                        C := evalhf(C + C);
                        base := evalhf(A + C);
                        base := floor(evalhf(base - A));
end do:
print(`The number base is` = base);
```

The number base is = 2 **(3.1.1)**

The machine epsilon is *the distance from 1 tot the next larger floating-point number*.

```
> eps := one:
while evalhf(one + eps - one) <> zero do
   eps := evalhf(eps/base);
end do:
eps := evalhf(eps*base):
print(`The machine epsilon is` = eps);
print(`The machine epsilon in base 2 is` = log[base](eps));
```

The machine epsilon is = $2.22044604925031308\ 10^{-16}$

The machine epsilon in base 2 is = -52.00000000 **(3.1.2)**

Floating-point arithmetic in Maple

```
> one := evalf(1):
zero := evalf(0):
A := one:
B := one:
while evalhf(B-one) = zero do
                        A := evalf(A + A);
                        B := evalf(A + one);
                        B := evalf(B - A);
end do:
base := 0:
C := one:
while base = 0 do
                        C := evalf(C + C);
                        base := evalf(A + C);
                        base := floor(evalf(base - A));
end do:
print(`The number base is` = base);
```

The number base is = 10 **(3.2.1)**

```
> eps := one:
while evalf(one + eps - one) <> zero do
   eps := evalf(eps/base);
end do:
eps := evalf(eps*base):
print(`The machine epsilon is` = eps);
print(`The machine epsilon in base 10 is` = log[base](eps));
print(`The number of digits Maple uses is` = Digits);
```

The machine epsilon is = $1.000000000\ 10^{-9}$

The machine epsilon in base 10 is = -9.000000000

The number of digits Maple uses is = 10 **(3.2.2)**

1.1. The golden ratio and roots of quadratic polynomials

Exercise: Compute the roots of the polynomial $0.001\,x^2 + 1000000\,x + 0.01$. Do you obtain the results you expected? How can you solve this problem?

We can obtain the roots of a quadratic polynomial with the discriminant formulas.

```
> s := solve(a*x^2+b*x+c=0, x);
```

$$s := \frac{-b+\sqrt{-4\,a\,c+b^2}}{2\,a}, -\frac{b+\sqrt{-4\,a\,c+b^2}}{2\,a} \tag{4.1}$$

```
> a := 10^(-3);
  b := 10^6;
  c := 10^(-2);
  r := solve(a*x^2+b*x+c=0, x);
  r1 := r[1]:
  r2 := r[2]:
```

$$a := \frac{1}{1000}$$

$$b := 1000000$$

$$c := \frac{1}{100}$$

$$r := -500000000 + \sqrt{249999999999999990}, -500000000 - \sqrt{249999999999999990} \tag{4.2}$$

```
> Digits := 50;
  r1_d50 := evalf(r1);
  r2_d50 := evalf(r2);
```

$$Digits := 50$$

$$r1\_d50 := -1.0000000000000000100000000000000000\ 10^{-8}$$

$$r2\_d50 := -9.9999999999999989999999999999999990000000000000000\ 10^{8} \tag{4.3}$$

```
> r1_hf := evalhf( (-b+sqrt(b^2-4*a*c))/(2*a));
  r2_hf := evalhf( -(b+sqrt(b^2-4*a*c))/(2*a));
```

$$r1\_hf := 0.$$

$$r2\_hf := -1.000000000\ 10^{9} \tag{4.4}$$

```
> evalf(abs(r1_hf-r1)/abs(r1));
  evalf(abs(r2_hf-r2)/abs(r2));
```

$$1.0000000000000000000000000000000000000000000000000$$

$$1.0000000000000000200000000000000003000000000000000\ 10^{-17} \tag{4.5}$$

We can use the product rule: the product of the two roots is the ratio of the constant term and the leading coefficient.

```
> expand(s[1]*s[2]);
```

```
c/a;
```

$$10$$

$$10 \tag{4.6}$$

```
> r1_hf_ver2 := evalhf(c/a/r2_hf);
```

$$r1\_hf\_ver2 := -1.0000000000000002\ 10^{-8} \tag{4.7}$$

```
> evalf(abs(r1_hf_ver2-r1)/abs(r1));
```

$$9.9999999999999990000000000000000010000000000000000\ 10^{-18} \tag{4.8}$$

1.2. Fibonnaci numbers, sequences and series

Exercise: What is the limit of the series $\sum_{n=1}^{\infty} \frac{1}{n^2}$? What is the result you find using floating-point arithmetic? Why do we find this result and how can we obtain a higher accuracy?

```
> Digits := 5:
  S := sum(1/(i^2),i=1..infinity);
  evalf(S);
```

$$S := \frac{\pi^2}{6}$$

$$1.6450 \tag{5.1}$$

We can't compute the sum of an infinite amount of numbers, hence we have to truncate the sum somewhere.

```
> n := 1:
  SUM := evalf(0):
  while evalf(SUM + 1/(n^2)) > SUM do
     SUM := evalf(SUM + 1/(n^2));
     n := n+1;
  end do:
  N := n:
  print(`The sum is` = SUM);
  abs(SUM-S)/abs(S);
  100*abs(SUM-S)/abs(S);
  print(`The number of terms is` = n);
```

$$The\ sum\ is = 1.6390$$

$$0.0036475$$

$$0.36475$$

$$The\ number\ of\ terms\ is = 142 \tag{5.2}$$

Catastrophic cancellation occured! We tried to add a very small number to a (relatively) larger one.

```
> evalf(1/(N^2));
  evalf(SUM + 1/(N^2) - SUM);
  evalf(SUM - SUM + 1/(N^2));
```

0.000049593

0.

0.000049593 **(5.3)**

We can solve this problem by computing the sum backwards: first adding all the small numbers together and working our way up.

```
> n := 12000:
  SUM := evalf(0):
  while n > 0 do
     SUM := evalf(SUM + 1/(n^2));
     n := n-1;
  end do:
  print(`The sum is` = SUM);
  abs(SUM-S)/abs(S);
  100*abs(SUM-S)/abs(S);
```

The sum is = 1.6449

0.000060792

0.0060792 **(5.4)**