11 Floating-Point Numbers

In the next chapters, floating-point numbers are at the heart of all computations. It is necessary to study them, as their behaviour follows precise rules.

How can we represent real numbers in a computer? In general, these numbers cannot be coded with a finite amount of information, and thus they cannot be exactly represented. It is necessary to approximate them using a finite amount of memory.

A standard has appeared around an approximation of real numbers with a finite quantity of information: the floating-point representation.

In this chapter, one will find: a basic description of the floating-point numbers and of the different kinds of these numbers available in Sage, and a demonstration of some of their properties. Examples will show some difficulties we encounter when computing with floating-point numbers and some tricks to get around them. We hope that the reader will develop a necessary careful approach. To conclude, we will try to describe some properties which must be fulfilled by numerical methods when they use these numbers.

To go further, the reader should refer to [BZ10] and [Gol91] (available on the internet) or to the book [MBdD+10].

11.1 Introduction

11.1.1 Definition

A set $F(\beta, r, m, M)$ of floating-point numbers is defined by four parameters: a *radix* $\beta \geq 2$, a number r of digits and two signed integers m and M. The elements of $F(\beta, r, m, M)$ are numbers of the form

$$x = (-1)^s \ 0.d_1 d_2 \dots d_r \cdot \beta^j,$$

where the digits d_i are integers verifying $0 \leq d_i < \beta$ for $i > 1$ and $0 < d_1 < \beta$. The amount r of digits is the *precision*: the sign s is 0 or 1; the *exponent* j lies in the range $[m, M]$, and $0.d_1 d_2 \ldots d_r$ is the *significand*.

11.1.2 Properties and Examples

The normalisation $0 < d_1 < \beta$ ensures that all floating-point numbers have the same amount of significant digits. One remarks that, with the convention $d_1 > 0$, the "zero" value cannot be represented: zero has a special representation.

As example, the number denoted by -0.028 in radix 10 (fixed-point representation) will be represented by $-0.28 \cdot 10^{-1}$ (assuming $r \geq 2$ and $m \leq -1 \leq M$). As the radix 2 is well adapted to the binary representation of the computers, we will always have $\beta = 2$ in the different sets of floating-point numbers proposed by Sage, and we will always use this setting in the remainder of this chapter. To give an example, $0.101 \cdot 2^1$ represents the value $5/4$ in the set $F(2, 3, -1, 2)$.

As the only possible value for d_1 when $\beta = 2$ is $d_1 = 1$, d_1 can be omitted in the machine implementation; considering again the set $F(2, 3, -1, 2)$, $5/4$ can be represented in the computer by the 5 bits: 00110, where the leftmost bit represents the + sign, the 2 following bits (01) represent the significand (101), and the last 2 ones at the right represent the exponent (00 encoding the value -1 of the exponent, 01 encoding 0, and so on).

It should be obvious for the reader that the sets $F(\beta, r, m, M)$ only describe a subset of the real numbers. To represent a real number x located between two consecutive numbers in $F(\beta, r, m, M)$, we need a function called *rounding* which will define which number will approximate x: for this we can use the nearest number from x, but other choices are available. The standard imposes that $F(\beta, r, m, M)$ is invariant by the rounding application. The set of numbers which can be represented is bounded, and the floating-point numbers contain the special values $+\infty$, $-\infty$ which represent the infinities (as $1/0$) but also all values greater than the largest positive number which can be represented (or less than the smallest negative number available), and also a representation of indefinite operations like $0/0$.

11.1.3 Standardisation

After some years of trials and errors, the need for a standard did arise, so that identical programs give the same results on different machines. Since 1985, the IEEE-754 standard defines different sets of numbers; among them the 64-bit "double-precision" numbers: the sign s is encoded on 1 bit, the significand on 53 bits (from which only 52 are stored), and the exponent on 11 bits. Numbers are of the form

$$(-1)^s \; 0.d_1 d_2 \ldots d_{53} \cdot 2^{j-1023}.$$

They correspond to the "double" type of the C programming language.

11.2 The Floating-Point Numbers

Sage provides two sorts of floating-point numbers:

1. the "double-precision" numbers as described in §11.1.3: these numbers are provided by the computer's processor; in Sage, they belong to the class `RDF`:

   ```
   sage: xrdf = RDF(3.0)
   ```

2. floating-point numbers with an arbitrary precision: every instance of the class `RealField` — or `Reals` — defines a set of floating-point numbers with a given precision (and possibly with a given rounding mode: see §11.3.2). For example, to declare a number `x100` with a precision on 100 binary digits, one writes:

   ```
   sage: R100 = RealField(100) # precision: 100 bits.
   sage: x100 = R100(3/8); x100
   0.37500000000000000000000000000
   ```

In the set `RealField(p)` numbers are of the form

$$(-1)^s \ 0.d_1 d_2 \dots d_p \cdot 2^e,$$

with $s \in \{0, 1\}$; the significand has p binary digits and e might have 30 binary digits (or more on some computers). An implicit precision is available:

```
sage: Rdefault = RealField()  # default precision of 53 bits
sage: xdefault = Rdefault(2/3)
```

and it is possible to check the precision of all floating-point numbers using the `prec()` method:

```
sage: xrdf.prec()
53
sage: x100.prec()
100
sage: xdefault.prec()
53
```

So, the numbers of the set `RealField()` and those of the set `RDF` have the same precision, but `RealField()` allows much larger exponents. The set `RealField()`, with a precision of 53 bits, is the default type of "real" numbers in Sage:

```
sage: x = 1.0; type(x)
<type 'sage.rings.real_mpfr.RealLiteral'>
sage: x.prec()
53
```

Here, `real_mpfr.RealLiteral` means that the set of numbers to which `x` belongs is implemented by the GNU MPFR library. Let us recall that the type of a variable is automatically defined by the right-hand side in an assignment:

```
sage: x = 1.0         # x belongs to RealField()
sage: x = 0.1e+1      # idem: x belongs to RealField()
sage: x = 1           # x is an integer
sage: x = RDF(1)      # x is a machine double-precision number
sage: x = RDF(1.)     # idem: x is a machine double-precision number
sage: x = RDF(0.1e+1) # idem
sage: x = 4/3         # x is a rational number
sage: R = RealField(20)
sage: x = R(1)        # x is a 20-bit floating-point number
```

and natural conversions from rational numbers are carried out:

```
sage: RDF(8/3)
2.6666666666666665
sage: R100 = RealField(100); R100(8/3)
2.6666666666666666666666666667
```

like conversions between different sets of floating-point numbers:

```
sage: x = R100(8/3)
sage: R = RealField(); R(x)
2.66666666666667
sage: RDF(x)
2.6666666666666665
```

The different sets of floating-point numbers contain the special values `+0`, `-0`, `+infinity`, `-infinity`, and[1] `NaN`:

```
sage: 1.0/0.0
+infinity
sage: RDF(1)/RDF(0)
+infinity
sage: RDF(-1.0)/RDF(0.)
-infinity
```

The special value `NaN` stands for undefined results:

```
sage: 0.0/0.0
NaN
sage: RDF(0.0)/RDF(0.0)
NaN
```

11.2.1 Which Kind of Floating-Point Numbers to Choose?

The arbitrary precision floating-point numbers allow us to compute with a very large precision, whereas the precision is fixed for `RDF` numbers. Computations with the `RealField(n)` numbers use the GNU MPFR software library, while for `RDF` numbers computations are carried out using the floating-point arithmetic of the processor, which is much faster. In §13.2.10 we give a comparison where the

[1] These results follow the IEEE-754 standard.

x	R2(x).ulp()	RDF(x).ulp()	R100(x).ulp()
10^{-30}	3.9e-31	1.75162308041e-46	1.24460305557222834142881281 08e-60
10^{-10}	2.9e-11	1.29246970711e-26	9.18354961579912115600575419 70e-41
10^{-3}	0.00049	2.16840434497e-19	1.54074395550978868244478235 41e-33
1	0.50	2.22044604925e-16	1.57772181044202361082345713 06e-30
10^{3}	510.	1.13686837722e-13	8.07793566946316088741610050 85e-28
10^{10}	4.3e9	1.90734863281e-06	1.35525271560688054250931600 11e-20
10^{30}	3.2e29	1.40737488355e+14	1.00000000000000000000000000

TABLE 11.1 – Distance between floating-point numbers.

efficiency of the processor's floating-point arithmetic is combined with libraries optimised for these numbers. Note that, among the numerical methods we will encounter in the next chapters, most of them only use `RDF` numbers and, whatever we do, a conversion of floating-point numbers to this set will occur.

R2, a Toy Set of Floating-Point Numbers. Arbitrary precision floating-point numbers, apart from being mandatory for large precision computations, enable us to define a class of floating-point numbers which, as they have very low accuracy, demonstrate in an extremal way the properties of floating-point numbers; the set `R2` of numbers with a precision of 2 bits:

```
sage: R2 = RealField(2)
```

11.3 Some Properties of Floating-Point Numbers

11.3.1 These Sets are Full of Gaps

In every set of floating-point numbers, the `ulp()` method (*unit in the last place*) returns the distance from a representable number to the next representable one (in the opposite direction from zero):

```
sage: x2 = R2(1.); x2.ulp()
0.50
sage: xr = 1.; xr.ulp()
2.22044604925031e-16
```

The reader can easily check the value given by `x2.ulp()`.

Table 11.1 gives the size of the interval which separates a given number x — or more exactly $R(x)$ where R is the considered set — from its nearest neighbour (in the opposite direction from zero) for different sets of numbers (`R100` is the set `RealField(100)`), and different values of x.

As expected, the size of the *gaps* between two consecutive numbers grows with the magnitude of the numbers.

Exercise 41 (a somewhat surprising value)**.** Show that `R100(`10^{30}`).ulp()` is exactly `1.00000000000000000000000000000`.

11.3.2 Rounding

How to approach a number which cannot be represented exactly in a set of floating-point numbers? There exist different possibilities to define *rounding*:

- in the direction of the nearest representable number: this is what is done in the set `RDF`, and it is the default behaviour of the sets created by `RealField`. For a number exactly in the middle of two representable numbers, rounding is done at the nearest even significand;
- in the direction of $-\infty$; for this, use `RealField(p,rnd='RNDD')` to obtain this behaviour with a precision of p bits;
- in the direction of zero: `RealField(p,rnd='RNDZ')`;
- in the direction of $+\infty$: `RealField(p,rnd='RNDU')`.

11.3.3 Some Properties

Rounding, which is necessary for the sets of floating-point numbers, gives rise to many unexpected effects. Let us explore some of them:

A Dangerous Phenomenon. Known as *catastrophic cancellation* it is the loss of precision which results from the subtraction of two very close numbers; more exactly, it is an amplification of the errors:

```
sage: a = 10000.0; b = 9999.5; c = 0.1; c
0.100000000000000
sage: a1 = a+c # add a small perturbation to a.
sage: a1-b
0.600000000000364
```

Here, the error `c` introduced on `a` makes the computation imprecise (the last 3 digits are false).

Application: Roots of a Quadratic Equation. Even computing the roots of a second-order equation can cause problems. Let us consider the case $a = 1$, $b = 10^4$, $c = 1$:

```
sage: a = 1.0; b = 10.0^4; c = 1.0
sage: delta = b^2-4*a*c
sage: x = (-b-sqrt(delta))/(2*a); y = (-b+sqrt(delta))/(2*a)
sage: x, y
(-9999.99990000000, -0.000100000001111766)
```

The sum of the roots is right, but not their product:

```
sage: x+y+b/a
0.000000000000000
sage: x*y-c/a
1.11766307320238e-9
```

The error is due to the phenomenon known as *catastrophic cancellation* which appears when we add `-b` and `sqrt(delta)` to compute `y`. Here, we can try to find a better approximation for `y`:

```
sage: y = (c/a)/x; y
-0.000100000001000000
sage: x+y+b/a
0.000000000000000
sage: x*y-c/a
-1.11022302462516e-16
```

We can remark that, due to rounding, the sum of the roots remains correct, but the product is much closer to c/a. The reader can consider all different choices for `a`, `b` and `c` to be convinced that writing a numerically robust program to compute the roots of a quadratic trinomial is far from easy.

The Set of Floating-Point Numbers is not an Additive Group. Actually, addition is not associative. Let us use the set `R2` (with 2 bits of precision):

```
sage: x1 = R2(1/2); x2 = R2(4); x3 = R2(-4)
sage: x1, x2, x3
(0.50, 4.0, -4.0)
sage: x1+(x2+x3)
0.50
sage: (x1+x2)+x3
0.00
```

We can deduce that different orders of computations in a program have some importance on the result!

Recurrences and Sequences of Floating-Point Numbers. Let us consider[2] the recurrence $u_{n+1} = 4u_n - 1$. If $u_0 = 1/3$, the sequence is stationary: $u_i = 1/3$ for all i.

```
sage: x = RDF(1/3)
sage: for i in range(1,100): x = 4*x-1; print(x)
0.333333333333
0.333333333333
0.333333333333
...
-1.0
-5.0
-21.0
-85.0
-341.0
-1365.0
-5461.0
```

[2]Thanks to Marc Deléglise (Institut Camille Jordan, Lyon, France) for this example.

```
-21845.0
...
```

The computed sequence is diverging! We can observe that this behaviour is natural, as it is a classical instability phenomenon: every error on u_0 is multiplied by 4 at every iteration, and we know that floating-point arithmetic introduces rounding errors, which will be amplified at every iteration.

Now, let us compute the recurrence $u_{n+1} = 3u_n - 1$, with $u_0 = 1/2$. We expect the same problem: the sequence is constant if computed exactly, but every error will be amplified at every iteration.

```
sage: x = RDF(1/2)
sage: for i in range(1,100): x = 3*x-1; print(x)
0.5
0.5
0.5
...
0.5
```

Now, the computed sequence remains constant! How can we explain these two different behaviours? Let us look at the binary representation of u_0 in both cases.

For the first case ($u_{n+1} = 4u_n - 1$, $u_0 = 1/3$), we have:

$$\frac{1}{3} = \frac{1}{4}\sum_{i=0}^{\infty}\frac{1}{4^i} = \frac{1}{4}\sum_{i=0}^{\infty}\frac{1}{2^{2i}},$$

and therefore 1/3 cannot be represented exactly in the set of floating-point numbers we have at our disposal. The reader of this book is invited to repeat the preceding computation in a large precision set like for example `RealField(1000)` to verify that the computed sequence is always diverging. Let us remark that if, in the first program, we replace the line

```
sage: x = RDF(1/3)
```

by

```
sage: x = 1/3
```

then the computations are carried out in the rational numbers and the iterates will remain equal to 1/3. In the second case ($u_{n+1} = 3u_n - 1$, $u_0 = 1/2$), u_0 and 3/2 in radix 2 are respectively 0.1 and 1.1; therefore they are exactly represented, without rounding, in the different sets of floating-point numbers: computation is exact, and the sequence remains constant.

The following exercise shows that a sequence encoded in a set of floating-point numbers may converge to a wrong limit.

Exercise 42 (an example of Jean-Michel Muller)**.** We consider the sequence (cf. [MBdD$^+$10, p. 9]):

$$u_n = 111 - \frac{1130}{u_{n-1}} + \frac{3000}{u_{n-1}u_{n-2}}.$$

It is possible to show that the general solution is of the form:

$$u_n = \frac{\alpha\ 100^{n+1} + \beta\ 6^{n+1} + \gamma\ 5^{n+1}}{\alpha\ 100^n + \beta\ 6^n + \gamma\ 5^n}.$$

1. Choose $u_0 = 2$ and $u_1 = -4$: what are the values of α, β and γ? To which limit is the sequence converging?
2. Write a program which computes the sequence (still with $u_0 = 2$ and $u_1 = -4$) in the set `RealField()` (or in `RDF`). What can we observe?
3. Explain this behaviour.
4. Carry out the same computation with a large precision, in `RealField(5000)` for example. Comment the result.
5. The recurrence is now defined in $\mathbb{Q}$. Program it in the set of rational numbers and comment the result.

The Summation of Numerical Series. We consider a numerical series with a positive general term u_n. The computation of partial sums $\sum_{i=0}^{m} u_i$ in a set of floating-point numbers is perturbed by rounding errors. The reader can enjoy showing that, if u_n tends to 0 when n tends to infinity, and if the partial sums remain in the interval of representable numbers, then after a certain rank m, the sequence $\sum_{i=0}^{m} u_i$ *computed with rounding* is stationary. In short, in the world of floating-point numbers, life is simple: series with positive general terms tending to 0 converge, provided the partial sums do not grow too much!

For example, let us look at the harmonic (diverging) series, with general term $u_n = 1/n$:

```
sage: def sumharmo(p):
....:     RFP = RealField(p)
....:     y = RFP(1.); x = RFP(0.); n = 1
....:     while x <> y:
....:         y = x; x += 1/n; n += 1
....:     return p, n, x
```

Let us test this function with different values for the precision `p`:

```
sage: sumharmo(2)
(2, 5, 2.0)
sage: sumharmo(20)
(20, 131073, 12.631)
```

The reader can verify using a sheet of paper and a pencil that, in our toy set `R2` of floating-point numbers, the function converges in 5 iterations to the value 2.0. Obviously, the result depends on the precision p, and the reader can also verify (always with a pencil...) that for $n > \beta^p$, the computed sum is stationary. However, be careful! With the default precision of 53 bits and doing 10^9 operations per second, it might need $2^{53}/10^9/3600$ hours, that is about 104 days, to reach the stationary value!

Improving the Computation of some Recurrences. With some care, it is possible to improve some results: here is a useful example.

It is common to encounter recurrences of the form:

$$y_{n+1} = y_n + \delta_n,$$

where the numbers δ_n have a small absolute value when compared to y_n: think for instance of the integration of the celestial mechanics ordinary differential equations to simulate the solar system: large values (of the distances, of the velocities) undergo very small perturbations in the long time [HLW02]. Even if it is possible to compute precisely the δ_n terms, rounding errors when doing the additions $y_{n+1} = y_n + \delta_n$ will introduce important errors.

As an example, consider the sequence defined by $y_0 = 10^{13}$, $\delta_0 = 1$ and $\delta_{n+1} = a\delta_n$ with $a = 1 - 10^{-8}$. The standard, naive, programming way to compute y_n is:

```
sage: def iter(y,delta,a,n):
....:     for i in range(0,n):
....:         y += delta
....:         delta *= a
....:     return y
```

As we have chosen rational values for y_0, δ_0 and a, we can compute the exact value of the iterates with Sage:

```
sage: def exact(y,delta,a,n):
....:     return y+delta*(1-a^n)/(1-a)
```

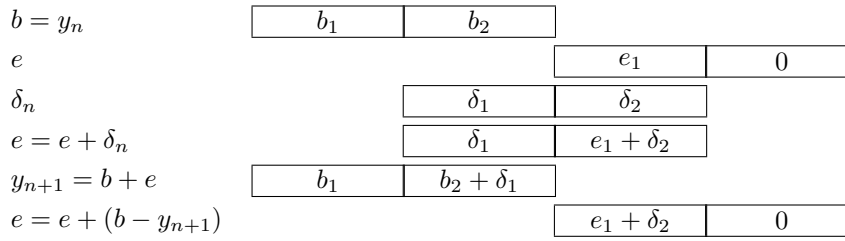
Now, let us compute again 100 000 iterates but with floating-point numbers in `RDF` (for instance), and let us compare the result with the exact value:

```
sage: y0 = RDF(10^13); delta0 = RDF(1); a = RDF(1-10^(-8)); n = 100000
sage: ii = iter(y0,delta0,a,n)
sage: s = exact(10^13,1,1-10^(-8),n)
sage: print("exact - classical summation: %.1f" % (s-ii))
exact - classical summation: -45.5
```

Now, this is the *compensated summation* algorithm:

```
sage: def sumcomp(y,delta,e,n,a):
....:     for i in range(0,n):
....:         b = y
....:         e += delta
....:         y = b+e
....:         e += (b-y)
....:         delta = a*delta # new value of delta
....:     return y
```

To understand the behaviour of this algorithm, let us look at the diagram below (we follow here the presentations of [Hig93] and [HLW02]), where the boxes represent the significand of the numbers. The position of the boxes represent the exponent (the more a box is to the left, the larger its exponent is):



The rounding error accumulates in e, and none of the digits of δ_n is lost, while with the naive method, the digits denoted δ_2 disappear from the computation.

Let us compute again the first 100 000 iterates with the compensated summation method:

```
sage: c = sumcomp(y0,delta0,RDF(0.0),n,a)
sage: print("exact - compensated summation: %.5f" \
           % RDF(s-RR(c).exact_rational()))
exact - compensated summation: -0.00042
```

The absolute error is -45.5 with the naive algorithm, and -0.00042 with the compensated summation! It should also be noted that the relative errors are respectively $4.55 \cdot 10^{-12}$ with the naive method and $4.16 \cdot 10^{-17}$ with the compensated summation.

11.3.4 Complex Floating-Point Numbers

Sage offers two families of complex numbers represented in the computer by pairs of floating-point numbers belonging to the sets we have encountered above:

1. double-precision complex numbers `ComplexDoubleField` (abbreviated `CDF`). These are numbers of the form $x + i \cdot y$ where x and y are both "double-precision" floating-point numbers. They are created like this:

   ```
   sage: x = CDF(2,1.); x
   2.0 + 1.0*I
   sage: y = CDF(20,0); y
   20.0
   ```

   or:

   ```
   sage: z = ComplexDoubleElement(2.,1.); z
   2.0 + 1.0*I
   ```

2. arbitrary precision complex numbers `ComplexField` — or `Complexes`. These are numbers of the form $x + i \cdot y$, where x and y have the same precision of p bits. An instance of the class `ComplexField` creates a set of given precision (53 by default):

   ```
   sage: C = ComplexField(); C(2,3)
   2.00000000000000 + 3.00000000000000*I
   sage: C100 = ComplexField(100); C100(2,3)
   2.0000000000000000000000000000 + 3.0000000000000000000000000000*I
   ```

Of course, computations with these sets face the same rounding problems as with real floating-point numbers.

11.3.5 Methods

We have already seen the `prec` and `ulp` methods. The different sets of numbers we have encountered provide a large amount of methods. Let us give some examples:

- methods which return constants. Examples:

```
sage: R200 = RealField(200); R200.pi()
3.1415926535897932384626433832795028841971693993751058209749
sage: R200.euler_constant()
0.57721566490153286060651209008240243104215933593992359880577
```

- trigonometric functions `sin`, `cos`, `arcsin`, `arccos`, and so on. Example:

```
sage: x = RDF.pi()/2; x.cos() # floating-point approximation of zero!
6.123233995736757e-17
sage: x.cos().arccos() - x
0.0
```

- the logarithms (`log`, `log10`, `log2`, etc.), the hyperbolic functions and their inverses (`sinh`, `arcsinh`, `cosh`, `arccosh`, etc.).

- special functions (`gamma`, `j0`, `j1`, `jn(k)`, and so on).

The reader should look at the Sage documentation to get a complete list of the very large number of available methods. We recall that this list can be obtained in the following way:

```
sage: x = 1.0; x.<tab>
```

For each method, we can get the parameters — if any — and an example of use by typing (here, for the Euler $\Gamma(x)$ function):

```
sage: x.gamma?
```

11.4 Interval and Ball Arithmetic

From what we explained above in this chapter, it should be clear that floating-point numbers only allow us to compute *approximations* of numerical results, and cannot provide proofs. But it turns out that we can compute rigorous *enclosures* of the sought quantities. For example, we generally cannot *prove*, computing with floating-point numbers, that a real number x_0 is the solution of some equation $f(x) = 0$ but we can *prove* that a solution exists in an interval $I = [\underline{x}, \overline{x}]$ or prove that no solution exists in I.

The tools for this are *interval arithmetic* and *ball arithmetic.*

- Given a set $\mathcal{F}$ of floating-point numbers, real interval (or inf-sup) arithmetic computes with intervals $\mathbf{a} = [\underline{a}, \overline{a}] = \{x \in \mathbb{R} : \underline{a} \leq x \leq \overline{a}\}$ (with $\underline{a}$ and $\overline{a} \in \mathcal{F}$). In the following we use bold letters to define an interval, and use the convention that any number $x \in \mathcal{F}$ is represented by the singleton $\mathbf{x} = [x, x]$.
- Real ball arithmetic (or mid-rad interval arithmetic): here we consider numbers $x \in \mathcal{F}$ and an error bound attached to x; balls are sets of the form $\{y \in \mathbb{R} : x - \varepsilon \leq y \leq x + \varepsilon\}$. We will also use bold letters to denote them.

As we will see later these families of sets can be extended to the complex plane.

We just give here a very minimal introduction to this important field; the interested reader should consult [Tuc11] for example (from which we adopt the notations).

Given real valued intervals (or real balls) $\mathbf{a}$ and $\mathbf{b}$, the *four operations* are defined in real valued intervals (or real balls) by:

$$\mathbf{a} * \mathbf{b} = [a * b \text{ for } a \in \mathbf{a} \text{ and } b \in \mathbf{b}],$$

where $*$ stands for one of $+$, $-$, $\times$, $\div$.

Observe that $\mathbf{a} \div \mathbf{b}$ is not yet defined if $0 \in \mathbf{b}$. To get around this, it is necessary to extend $\mathcal{F}$ with the infinite values $\pm\infty$, and to consider infinite intervals in one or both directions. The following table of examples shows the results for all cases where $0 \in \mathbf{b}$, when computing with a class of real intervals:

$\mathbf{a}$	$\mathbf{b}$	Remarks	$\mathbf{a} \div \mathbf{b}$
$[-1, +1]$	$[-2, +3]$	$0 \in \mathbf{a}$	`[-infinity .. +infinity]`
$[-2, -1]$	$[-1, 0]$	$\overline{a} < 0$	`[1.0000000000000000 .. +infinity]`
$[-2, -1]$	$[0, +2]$	$\overline{a} < 0$	`[-infinity .. -0.50000000000000000]`
$[+2, +3]$	$[-3, 0]$	$0 < \underline{a}$	`[-infinity .. -0.66666666666666662]`
$[+2, +3]$	$[0, +4]$	$\underline{a} > 0$	`[0.50000000000000000 .. +infinity]`
$[-2, -1]$	$[0]$	$0 \notin \mathbf{a}$	`[-infinity .. +infinity]`
$[0]$	$[0]$		`[.. NaN ..]`

In the following, we always assume that $\mathcal{F}$ contains $+\infty$ and $-\infty$. Note that the rounding mode must be chosen so as to guarantee the results: thus, when doing arithmetic operations on intervals, one must always round outwards the resulting interval.

11.4.1 Implementation in Sage

Recall that we have arbitrary precision real numbers in Sage (and that different rounding modes are available); then:

- `RealIntervalField(n)` is the set of real intervals with a precision of `n` bits. It is implemented using pairs of numbers from `RealField(n)`. In the default case (53 bits of precision) `RealIntervalField()` can be abbreviated by `RIF`. We can create a `RIF` (or a `RealIntervalField(n)`) object from a numeric expression:

```
sage: r3 = RIF(sqrt(3)); r3
1.732050807568877?
sage: print(r3.str(style="brackets"))
[1.7320508075688769 .. 1.7320508075688775]
```

As $\sqrt{3}$ cannot be exactly represented in `RealField()`, it is represented by an interval which contains the exact value. For printing, the default is to use the "question style", where the "known correct" part of the number, followed by a question mark is printed. The question mark indicates that the preceding digit is possibly wrong by at most ± 1. The "brackets" style prints outward approximations of the minimum and maximum of the interval (bounds are stored in binary: what is printed are decimal values of the bounds rounded in the direction of $-\infty$ and $+\infty$ respectively).

With the following instruction we fix the printing style to "brackets" (replace `brackets` by `question` to go back to the default style):

```
sage: sage.rings.real_mpfi.printing_style = 'brackets'
```

When we create a `RIF` object from a number which can be exactly represented in $\mathcal{F}$, the interval created is a singleton (an interval of diameter 0), as expected:

```
sage: r2 = RIF(2); r2, r2.diameter()
([2.0000000000000000 .. 2.0000000000000000], 0.000000000000000)
```

We can also create intervals of any size by giving two real numbers, and the result is the smallest representable interval which contains them:

```
sage: rpi = RIF(sqrt(2),pi); rpi
[1.4142135623730949 .. 3.1415926535897936]
sage: RIF(0,+infinity)
[0.00000000000000000 .. +infinity]
```

- `RealBallField(n)` is the set of real balls, with a precision of `n` bits. The default case (53 bits of precision) `RealBallField()` can be abbreviated by `RBF`. Some available constructions are:

```
sage: RBF(pi)
[3.141592653589793 +/- 5.61e-16]
sage: RealBallField(100)(pi)
[3.1415926535897932384626433832 +/- 3.83e-30]
```

Observe that, like for intervals, exactly representable numbers produce a ball of radius 0:

```
sage: RBF(2).rad()
0.00000000
```

Some Methods for Intervals and Balls. The following methods do not require further comment:

```
sage: si = sin(RIF(pi))
sage: si.contains_zero()
True
sage: sb = sin(RBF(pi))
sage: sb.contains_zero()
True
```

The standard functions exp, sin, cos, arcsin, arccos, sinh, cosh ... are all defined, and there exist methods to get the centre of an interval, the diameter of an interval or a ball, a bisection method which cuts an interval into two almost equal ones and so on. Note that the middle of an interval is not always in $\mathcal{F}$, and thus it is not always possible to cut an interval into two *equal* ones. Here is an extreme case where the left sub-interval is a singleton, and the right one is identical to the original interval:

```
sage: a = RealIntervalField(30)(1, RR(1).nextabove())
sage: a.bisection()
([1.0000000000 .. 1.0000000000], [1.0000000000 .. 1.0000000019])
```

For the same reason, the centre and diameter might be inexact:

```
sage: b = RealIntervalField(2)(-1,6)
sage: b.center(), b.diameter()
(2.0, 8.0)
```

It is also possible to build `RealBallField()` objects from `RealIntervalField()` intervals and conversely:

```
sage: s = RIF(1,2)
sage: b = RBF(s)
sage: bpi = RBF(pi)
sage: ipi = RIF(bpi)
```

But beware:

```
sage: RIF(RBF(RIF(1,2))) == RIF(1,2)
False
sage: RBF(RIF(RBF(pi))) == RBF(pi)
False
```

Why that? The reason lies in the different implementations of both classes:

- As already said above, `RealIntervalField(n)` objects are represented by pairs of `RealField(n)` numbers, and rely on the `MPFI` library [RR05].
- The implementation of `RealBallField(n)` objects is different: the ball midpoint and its radius are stored, but only the midpoint is stored in full-precision. The radius is represented by a floating-point number with fixed-precision significand and arbitrary-precision exponent. Generally, a few bits suffice for the radius. The implementation in Sage relies on the `Arb` library [Joh13].

Consequently, ball arithmetic is less computationally expensive than inf-sup interval arithmetic and even, at high precision, it is not more expensive than plain floating-point arithmetic.

11.4.2 Computing with Real Intervals and Real Balls

Once we have defined sets of intervals or balls, we must define functions operating on them. For a given interval (or a ball) $\mathbf{x}$, the range of a function f on $\mathbf{x}$ is given by $R(f, \mathbf{x}) = \{f(x) \text{ for } x \in \mathbf{x}\}$. For a continuous function f we just define its interval extension $\mathbf{F}(\mathbf{x})$ as the smallest interval (or ball) included in $\mathcal{F}$ which contains $R(f, \mathbf{x})$.

Examples:

```
sage: E = RIF(-pi/4,pi)
sage: sin(E)
[-0.70710678118654769 .. 1.0000000000000000]
sage: E = RIF(-1,2); exp(E)
[0.36787944117144227 .. 7.3890560989306505]
sage: E = RIF(0,1); log(E)
[-infinity .. -0.00000000000000000]
```

For a set of *standard functions* $(\exp, \sin, \cos, \arcsin, \arccos, \sinh, \cosh, \ldots)$, $\mathbf{F}(\mathbf{x})$ is computed as precisely as possible and is a tight approximation of $R(f, \mathbf{x})$.

But things become less obvious when we compose standard functions and combine them with arithmetic operators. Example:

```
sage: E=RIF(-pi,pi)
sage: f = lambda x: sin(x)/x
sage: f(E)
[-infinity .. +infinity]
```

A mathematician could expect to get the interval $[0, 1]$, extending $s(x) = \sin(x)/x$ by continuity to 1 in 0. But interval arithmetic and ball arithmetic are not substitutes for doing mathematical analysis! Actually for an interval $\mathbf{x}$, the interval $\mathbf{s} = \sin(\mathbf{x})$ is evaluated and then $\mathbf{s} \div \mathbf{x}$ is computed using the rules described above for the division; this explains the result we got: $\sin(x)/x$ does not belong to the set of *standard functions*.

We define the set $\mathcal{E}$ of *elementary functions* as the functions obtained by combining standard functions, constants and variables using arithmetic operations and composition. For example, consider $f(x) = (1 + x^2)\sin(2x + 1)$: this is an elementary function. To evaluate the extension $\mathbf{F}(\mathbf{x})$ of f for a given interval (or ball) $\mathbf{x}$, we evaluate $\mathbf{I}_1 = 2\mathbf{x} + 1$, $\mathbf{I}_2 = 1 + \mathbf{x}^2$, $\mathbf{I}_3 = \sin(\mathbf{I}_1)$ and then $\mathbf{F}(\mathbf{x}) = \mathbf{I}_2.\mathbf{I}_3$ (such a decomposition is generally not unique). From this it is not difficult to deduce that, for an elementary function f, we just have:

$$R(f, \mathbf{x}) \subseteq \mathbf{F}(\mathbf{x}).$$

Moreover, there often exist different possible extensions of a function to intervals: for example let us consider the real valued function $f_1(x) = 1 - x^2$; we

can also write it as $f_2(x) = 1 - x \cdot x$ or $f_3(x) = (1-x) \cdot (1+x)$. It turns out that the extension of these functions to intervals are different. The reader will verify by hand the following results:

```
sage: x = RIF(-1,1)
sage: 1-x^2
[0.00000000000000000 .. 1.0000000000000000]
sage: 1-x*x
[0.00000000000000000 .. 2.0000000000000000]
sage: (1-x)*(1+x)
[0.00000000000000000 .. 4.0000000000000000]
```

Exercise 43. Explain why the outputs of `1-x^2` and `1-x*x` differ.

11.4.3 Some Examples of Applications

Finding Roots by Bisection. Let $f : I \mapsto \mathbb{R}$ be a continuous elementary function. We want to find *all* the zeros of f in I. More precisely, we want to find (tiny) intervals such that their union contains all the zeros of f in I (and with only one root by interval). Recall that for an interval $\mathbf{x} \subset I$, we have $R(f, \mathbf{x}) \subseteq \mathbf{F}(\mathbf{x})$. Conversely, $y \notin \mathbf{F}(\mathbf{x}) \Rightarrow y \notin R(f, \mathbf{x})$. Then we get a simple algorithm: we recursively cut I into sub-intervals until a threshold size is attained; for any sub-interval $\mathbf{s}$, if $0 \notin \mathbf{F}(\mathbf{s})$, we can throw away $\mathbf{s}$ as we know that it cannot contain any root. Moreover, when the bisection is finished, if we know the derivative of f and if it is continuous, we can check that f is monotonic on all the computed intervals, and so obtain a *proof* of the existence of only one root by interval.

Example: find *all* the roots of $\sin(1/x)$ in the interval $[1/64, 1/32]$, with a precision of 100 bits (here, we compute with intervals):

```
sage: def bisect(funct,x,tol,zeros):
....:     if 0 in funct(x):
....:         if x.diameter()>tol:
....:             x1,x2 = x.bisection()
....:             bisect(funct,x1,tol,zeros)
....:             bisect(funct,x2,tol,zeros)
....:         else:
....:             zeros.append(x)
sage: sage.rings.real_mpfi.printing_style = 'question'
sage: fs = lambda x: sin(1/x)
sage: d = RealIntervalField(100)(1/64,1/32)
sage: zeros = []
sage: bisect(fs,d,10^(-25),zeros)
sage: for s in zeros:
....:     s
0.015915494309189533576888377?
0.016753151904410035344093 03?
0.017683882565766148418764 87?
```

```
0.018724110951987686561045148?
0.019894367886486916971110 47?
0.021220659078919378102517835?
0.022736420441699333681269 11?
0.024485375860291590118289809?
0.026525823848649222628147293?
0.028937262380344606503433 42?
sage: dfs = lambda x: -cos(1/x)/x^2
sage: not any([dfs(I).contains_zero() for I in zeros])
True
```

So, $\sin(1/x)$ has exactly 10 roots in the interval $[1/64, 1/32]$: this computation is a *proof*.

Note that Newton's method (see page 270) can be generalised to interval arithmetic [Tuc11].

Proving that a Matrix is not Singular. Let us consider the matrix M of size n with term $M_{i,j} = (1 + \log i)/(i^2 + j^2)$:

```
sage: def NearlySingularMatrix(R,n):
....:     M=matrix(R,n,n)
....:     for i in range(0,n):
....:         for j in range(0,n):
....:             M[i,j]= (1+log(R(1+i)))/((i+1)^2+(j+1)^2)
....:     return M
```

We fix $n = 35$. Let us build M in `RDF` and compute its determinant:

```
sage: n=35
sage: NearlySingularMatrix(RDF,n).det()
0.0
```

So, the determinant seems to be zero. Now we compute with `RBF` balls:

```
sage: NearlySingularMatrix(RBF,n).det().contains_zero()
True
```

We cannot conclude whether the matrix is singular or not, as the computed ball determinant contains zero. Let us compute with balls of growing precision:

```
sage: def tryDet(R,n):
....:     p = 53
....:     z = True
....:     while z:
....:         p += 100
....:         MRF=NearlySingularMatrix(R(p),n)
....:         d = MRF.det()
....:         z = d.contains_zero()
....:     return p,d
sage: tryDet(RealBallField,n)
```

```
(1653, [9.552323592707808e-485 +/- 1.65e-501])
```

For a precision of 1653 bits, we have found a ball $9.552323592707808 \cdot 10^{-485} \pm 1.65 \cdot 10^{-501}$ which contains the exact value of the determinant, and thus the determinant of M is not equal to zero. We have *proven* that the matrix is non singular (note that a more serious program should include, for safety, a limit to the loop on the precision `p`).

We can make the same computation with intervals:

```
sage: tryDet(RealIntervalField,n)
(1653, 9.552323592707808?e-485)
```

Here also we get the same conclusion: M is non singular. Let us compare the computing times:

```
sage: time p,d = tryDet(RealBallField,n)
CPU times: user 4.75 s, sys: 12 ms, total: 4.76 s
Wall time: 4.75 s
sage: time p,d = tryDet(RealIntervalField,n)
CPU times: user 6.62 s, sys: 8 ms, total: 6.63 s
Wall time: 6.62 s
```

As could be expected (1653 bits is a large precision!) `RealBallField` computations are less expensive than `RealIntervalField` ones.

11.4.4 Complex Intervals and Complex Balls

`ComplexIntervalField(n)` and `ComplexBallField(n)` define square boxes in the complex plane, with a precision of `n` bits. The default cases (53 bits of precision) can be called `CIF` and `CBF`. Constructors accept numerical complex quantities:

```
sage: CBF(sqrt(2),pi)
[1.414213562373095 +/- 4.10e-16] + [3.141592653589793 +/- 5.61e-16]*I
sage: CIF(sqrt(2),pi)
1.414213562373095? + 3.141592653589794?*I
sage: I = CIF(0,1)
sage: CIF(sqrt(2)+pi*I)
1.414213562373095? + 3.141592653589794?*I
sage: CBF(sqrt(2)+pi*I)
[1.414213562373095 +/- 4.10e-16] + [3.141592653589793 +/- 5.61e-16]*I
```

and real intervals or balls:

```
sage: c = CIF(RIF(1,2),RIF(-3,3))
sage: c.real()
[1.0000000000000000 .. 2.0000000000000000]
sage: c.imag()
[-3.0000000000000000 .. 3.0000000000000000]
sage: CBF(RIF(1,2),RIF(-3,3))
[+/- 2.01] + [+/- 3.01]*I
```

Standard functions are defined, and also methods to compute the argument, the norm and so on:

```
sage: ComplexIntervalField(100)(1+I*pi).arg()
1.2626272556789116834443220836 1?
sage: ComplexBallField(100)(1+I*pi).arg()
[1.2626272556789116834443220836 0 +/- 6.60e-30]
sage: ComplexIntervalField(100)(1+I*pi).norm()
10.869604401089358618834490999 9?
```

11.4.5 Usage and Limitations

First, not every computation can be carried out with balls or intervals. For example, finding the roots of a polynomial is not implemented:

```
sage: (x^3+2*x-1).roots(ring=RR)
[(0.453397651516404, 1)]
sage: (x^3+2*x-1).roots(ring=RBF)
...
NotImplementedError: root finding for this polynomial not implemented
```

(and the same happens with `ring=RIF`). Linear algebra also has some limitations: for example matrix inversion, computation of echelon form, minimal polynomial, solution of linear systems are available, but not eigenvalue computations.

Secondly, in case both `RealBallField` and `RealIntervalField` are available for a given computation, which one should we use? There is no definitive answer but the examples above show that when dealing with potentially large intervals, it is easier to use `RealIntervalField`, and that `RealBallField` is faster when computing with numbers tainted by errors.

11.4.6 Interval Arithmetic is Used by Sage

Internally, Sage uses interval arithmetic; for example Sage can compute in the field of algebraic numbers (roots of polynomials in $\mathbb{Z}[x]$). Numbers are represented by their minimal polynomial and computations are exact. But, how can we print the results? How can we get a human understandable result? Interval arithmetic is the answer (see also pages 140 and 275 of this book):

```
sage: x=QQbar(sqrt(3)); x
1.732050807568878?
sage: x.interval(RealIntervalField(100))
1.7320508075688772935274463415 1?
```

11.5 Conclusion

All numerical methods implemented in Sage, as those described in the next chapters, have been theoretically studied: this numerical analysis includes studying

Sets of floating-point numbers	
Machine floating-point numbers	`RDF`
Machine complex floating-point numbers	`CDF`
Real numbers with a precision of p bits	`RealField(p)`
Complex numbers with a precision of p bits	`ComplexField(p)`
Real intervals with a precision of p bits	`RealIntervalField(p)`
Complex intervals with a precision of p bits	`ComplexIntervalField(p)`
Real balls with a precision of p bits	`RealBallField(p)`
Complex balls with a precision of p bits	`ComplexBallField(p)`

TABLE 11.2 – A summary of floating-point classes.

the convergence of the iterative methods, the error introduced when simplifying a problem to make it computable, but also the behaviour of the computations in presence of perturbations such as those introduced by the inexact arithmetic of the floating-point numbers.

Let us consider an algorithm $\mathcal{F}$ which, from some data d, computes $x = \mathcal{F}(d)$. This algorithm can only be used if it does not increase the errors on d too much: to a perturbation ε of d corresponds a perturbed solution $x_\varepsilon = \mathcal{F}(d + \varepsilon)$. It is absolutely necessary that the error $x_\varepsilon - x$ introduced depends only moderately on ε (in a continuous way, does not grow too fast, ...): all numerical algorithms must have stability properties to be usable. In Chapter 13, we will explore the stability problems for the algorithms used in numerical linear algebra.

Let us also remark that some computations are definitively not possible in finite precision, like for example the sequence given in Exercise 42: every perturbation, however small, will lead the sequence to converge to a wrong value. This is a typical instability for the solution of a problem: the experimental study of a sequence with floating-point numbers should be performed with great care.

The reader might find that performing computations with floating-point numbers is hopeless, but this opinion should be moderated: an overwhelming part of available computing resources is used to perform computations in these sets of numbers: the approximate solution of partial differential equations, optimisation or signal processing, etc. Floating-point numbers should be used with care, but they did not prevent the development of computing and its applications: rounding errors do not limit the validity of numerical weather forecasts, to give only one obvious example.

Interval arithmetic seems to appear first in the works of Ramon E. Moore in 1966 [Moo66] (even if the idea appears before the computer era), but new software developments (such as `MPFI` and `Arb` used by Sage) as well as some spectacular applications draw attention to it: some famous old mathematical conjectures have proofs which rely at least partially on interval computations. For example, we may mention the Kepler conjecture in 1998 by Thomas C. Hales, or the ternary Goldbach problem by Harald A. Helfgott in 2013.