Exact and float point computations - Stability and unstability in dynamics

Exact computations and floating point approximations

There are a lot of different numbers in Sage. You will need to choose which kind of numbers you want to use depending on your computations. In this worksheet we will consider four types of numbers:

- rationals
- floating point numbers
- algebraic numbers

If you are interested in all possible ways of representing a real number in Sage, you can have a look at the tutorial "Real and complex numbers in Sage".

Floating point and rationals

To create an integer or a rational number, you just write it as you would do on a sheet of paper

```
sage: 2 + 3^5    # an integer
sage: 23 / 45    # a rational number
```

To create a floating point number, you need to add a dot

```
sage: 1.0
sage: 3.25 + 22.18
```

Contrarily to integers and rationals, a floating point number has limited precision

```
sage: 2^100 + 2^10 - 2^100
sage: 2.0^100.0 + 2.0^10.0 - 2.0^100.0
```

Because floating point numbers have limited precision they are much faster. In the following we will see that the the bad choice of numbers influence the computational time and the correctness.

Maps of the interval: fixed points and iteration

Let us consider the map $f_4(x) = 4x(1-x)$ from the interval $[0,1]$ to itself. Prove that f_4 is surjective and plot it:

```
sage: # edit here
```

Show that $f_4(3/4) = 3/4$ (in other words, the point $3/4$ is a fixed point of f_4). What do you expect from the following two commands (you have to guess whether the answer will be *True* or *False* and you can then check your answer by executing the cell):

```
sage: s = 3.0 / 4.0
sage: 4 * s * (1 - s) == s
```

Now let $f_{7/2}(x) = 7/2x(1-x)$. Prove that $5/7$ is a fixed point of $f_{7/2}$. Is the following *True* or *False*:

```
sage: s = 5.0 / 7.0
sage: 7.0 / 2.0 * s * (1.0 - s) == s
```

Perform the same two computations as above with rational numbers instead of floating point.:

```
sage: # edit here
```

On a computer a floating point number is a number of the form $m2^n$ where m (the mantissa) and n (the exponent) have some fixed bounds. In particular, floating point numbers have *finite* precision. Computations with floating points numbers are inaccurate but very efficient.

Compare the following computation with rationals:

```
sage: s = 1
sage: for i in range(10):
....:     s = (s + 2/s) / 2
sage: print s
sage: print s.numerical_approx()
```

and the same computation with floating point numbers:

```
sage: s = 1.0
sage: for i in range(10):
....:     s = (s + 2.0 / s) / 2.0
....: print s
```

What can you say?

Now compare the following computation with rationals:

```
sage: s = 5 / 7
sage: for i in range(100):
....:     s = 7 / 2 * s * (1 - s)
sage: print s
sage: print s.numerical_approx()
```

and the same computation with floating point numbers:

```
sage: s = 5.0 / 7.0
sage: for i in range(100):
....:     s = 7.0 / 2.0 * s * (1.0 - s)
sage: print s
```

What can you say? What is the difference between the two cases we considered above? What makes the functions $s \mapsto (s+2/s)/2$ and $s \mapsto 7/2s(1-s)$ different?