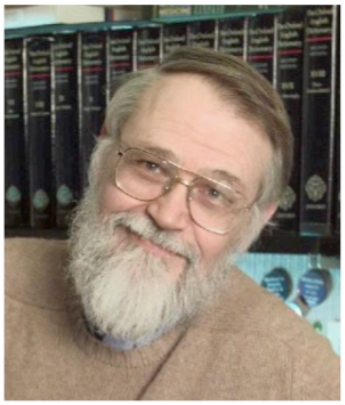
480: 05092008 -- floating point; interval arith

Floating Point Numbers and Interval Arithmetic

Are floating point numbers just broken?

> Floating point numbers are like piles of sand; every time you move them around, you lose a little sand and pick up a little dirt. - Kernighan and Plauger



(from http://www.cs.Princeton.EDU/IntroCS)

To a mathematician like me floating point numbers are so "broken" it isn't even funny.
Solution: Understand this and learn to live with it.

EXAMPLE: Rounding can be painful.

Make Sage use Python floats for all decimal literals!

```
RealNumber = float
```

Example

```
3.1415
```

```
3.1415000000000002
```

```
preparse('3.1415')
```

```
"RealNumber('3.1415')"
```

```
1/10
```

```
1/10
```

```
preparse('1/3')
```

```
'Integer(1)/Integer(3)'
```

```
if 0.1 + 0.2 == 0.3:
   print "YES (0.3)"
else:
   print "very weird (0.3)"
if 0.1 + 0.3 == 0.4:
   print "YES (0.4)"
else:
   print "very weird (0.4)"
```

```
very weird (0.3)
YES (0.4)
```

Are computers just broken?!

```
a = (1234.567 + 45.67844) + 0.0004
a
```

```
1280.2458399999998
```

```
b = 1234.567 + (45.67844 + 0.0004)
b
```

```
1280.24584
```

```
a == b
```

```
False
```

```

```

```
a = (1234.567 + 1.234567) * 3.333333
a
```

```
4119.3381447328111
```

```
b = 1234.567 * 3.333333 + 1.234567 * 3.333333
```

```
b
```

```
4119.338144732812
```

```
a == b
```

```
var('x,y')
expand((x - y) * (x + y))
```

```
x^2 - y^2
```

```
RealNumber = float
x = 1.0000001; y = 1
x^2 - y^2
```

```
2.0000001010878066e-07
```

```
(x-y)*(x+y)
```

```
2.0000001011677347e-07
```

```
(x^2 - y^2) - (x-y)*(x+y)
```

```
-7.9928146485283916e-18
```

```

```

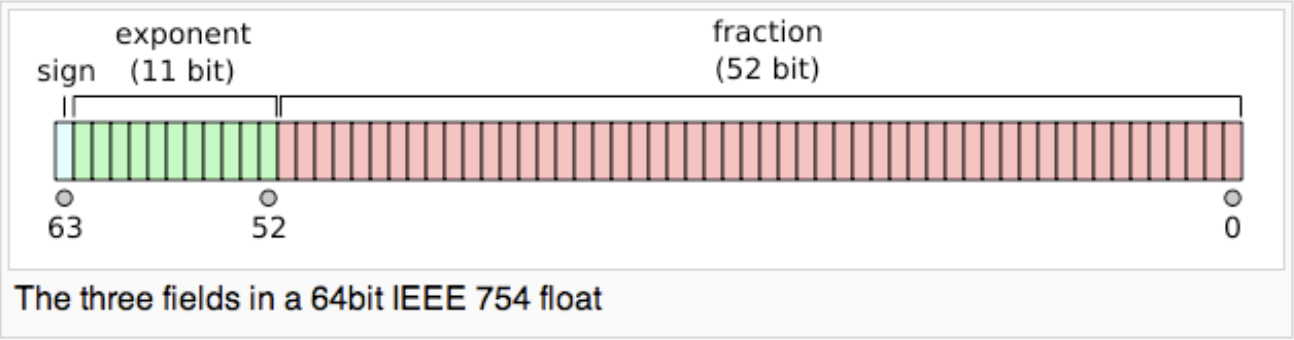
```

```

IEEE Standard 754-1985 Double Precision Arithmetic

Double Precision

The IEEE double precision floating point standard representation requires a 64 bit word, which may be represented as numbered from 0 to 63, left to right. The first bit is the sign bit, S, the next eleven bits are the exponent bits, 'E', and the final 52 bits are the fraction 'F':

```
S EEEEEEEEEEE FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
0 1         11 12                                                63
```



The three fields in a 64bit IEEE 754 float

The value V represented by the above 64-bits is determined as follows:

- If $E = 2047$ and $F \neq 0$, then V=NaN ("Not a number")
- If $E = 2047$ and $F = 0$ and $S = 1$, then $V = -\infty$
- If $E = 2047$ and $F = 0$ and $S = 0$, then $V = \infty$
- If $0 < 2047$ then

  $$V = (-1)^S \cdot 2^{E-1023} \cdot (1.F)$$

  where "1.F" represents the binary number created by prefixing F with an implicit leading 1 and a binary point. This is called "exponent biasing" and is done because exponents are signed, and for technical reasons this is the most efficient representation of exponents.
- If $E = 0$ and F is nonzero, then

  $$V = (-1)^S \cdot 2^{-1022} \cdot (0.F)$$

- If $E = 0$ and F is zero and S is 1, then $V = -0$
- If $E = 0$ and F is zero and S is 0, then $V = 0$

There are also rules for addition, multiplication, and division, etc., of such numbers with various rounding rules. All are implemented very quickly in hardware. There are also single precision floating point numbers, which are of course very useful in video games where one can't "see" the difference between single and double precision.
For more, see this page and this Wikipedia page.

EXERCISE (to make sure you're paying attention): In Python the maximum exponent is 309. Why isn't it 1023?

```
float(10^308)
```

```
9.9999999999999981e+307
```

```
float(10^309)
```

```
inf
```

```
2.0^1023
```

```
8.9884656743115795e+307
```

```
2.0^1024
```

```
Traceback (click to the left for traceback)
...
OverflowError: (34, 'Result too large')
```

```
%cython

def print_c_double(n):
```

```
    cdef double x
    x = n
    printf("%e\n",x)
```

__Users_wa..._code_sage236_spyx.c __Users_wa...age236_spyx.pyx.html

```
print_c_double(1.39993)
```

```
    1.399930e+00
```

```
1.39993
```

```
    1.3999299999999999
```

```
print_c_double(10^100)
```

```
    1.000000e+100
```

```
print_c_double(10^310)
```

```
    inf
```

```
print_c_double(-19^(-100))
```

```
    -1.332416e-128
```

```
(-19.)^(-100)
```

```
    1.3324162010917425e-128
```

```

```

```

```

Interval Arithmetic

Instead of doing arithmetic with floats do arithmetic instead with *intervals*. Then at least you know for certain *something*.

Multiplication of intervals:

$T \cdot S = \{ x \mid \text{there is some } y \text{ in } T, \text{ and some } z \text{ in } S, \text{ such that } x = y * z \}.$

The basic operations of interval arithmetic are, for two intervals $[a, b]$ and $[c, d]$ that are subsets of the real line $(-\infty, \infty)$,

- $[a,b] + [c,d] = [a + c, b + d]$
- $[a,b] - [c, d] = [a - d, b - c]$
- $[a,b] * [c,d] = [\min(ac, ad, bc, bd), \max(ac, ad, bc, bd)]$

- $[a,b] / [c,d] = [\min(a/c, a/d, b/c, b/d), \max(a/c, a/d, b/c, b/d)]$

Division by an interval containing zero is not defined under the basic interval arithmetic. The addition and multiplication operations are commutative, associative and sub-distributive: the set $X(Y + Z)$ is a subset of $XY + XZ$.

There are interval versions of standard algorithms, such as Newton's method.

```
# Make it so all float literals are intervals
RealNumber = RIF
```

```
RIF
```
```
    Real Interval Field with 53 bits of precision
```
```
restore('RealNumber')
```

```
1.2349
```
```
    1.23490000000000
```
```
a = RIF((0,0.1))
```

```
a
```
```
    [0.00000000000000000 .. 0.10000000000000001]
```
```
a^2
```
```
    [0.00000000000000000 .. 0.01000000000000002]
```
```
eps=0.1
RIF(5-eps, 5+eps)
```
```
    [4.9000000000000003 .. 5.0999999999999997]
```
```
0.1
```
```
    [0.099999999999999991 .. 0.10000000000000001]
```
```
a = 0.1
```

```
a.absolute_diameter()
```
```
    1.38777878078145e-17
```
```
a.relative_diameter()
```
```
    1.38777878078145e-16
```
```
(0.7).absolute_diameter()
```
```
    1.11022302462516e-16
```
```
(0.7).relative_diameter()
```
```
    1.58603289232165e-16
```

```
RealNumber = RIF

# Example
if 0.1 + 0.2 == 0.3:
   print "YES (0.3)"
else:
   print "very weird (0.3)"
if 0.1 + 0.3 == 0.4:
   print "YES (0.4)"
else:
   print "very weird (0.4)"
```

```
very weird (0.3)
very weird (0.4)
```

```

```

```
# Example
if (0.1 + 0.2).overlaps(0.3):
   print "YES (0.3)"
else:
   print "very weird (0.3)"
if (0.1 + 0.3).overlaps(0.4):
   print "YES (0.4)"
else:
   print "very weird (0.4)"
```

```
YES (0.3)
YES (0.4)
```

```
a = (1234.567 + 45.67844) + 0.0004
a
```

```
[1280.2458399999995 .. 1280.2458400000003]
```

```
b = 1234.567 + (45.67844 + 0.0004)
b
```

```
[1280.2458399999995 .. 1280.2458400000001]
```

```
a.overlaps(b)
```

```
True
```

```
a = (1234.567 + 1.234567) * 3.333333
a
```

```
[4119.3381447328092 .. 4119.3381447328120]
```

```
b = 1234.567 * 3.333333 + 1.234567 * 3.333333
b
```

```
[4119.3381447328092 .. 4119.3381447328120]
```

```
a.overlaps(b)
```

```
True
```

```
x = 1.0000001; y = 1
x^2 - y^2
```

```
    [2.0000000966469144e-7 .. 2.0000001033082527e-7]
```

```
(x-y)*(x+y)
```

```
    [2.0000000967268418e-7 .. 2.0000001011677348e-7]
```

```
a = (x^2 - y^2) - (x-y)*(x+y)
a
```

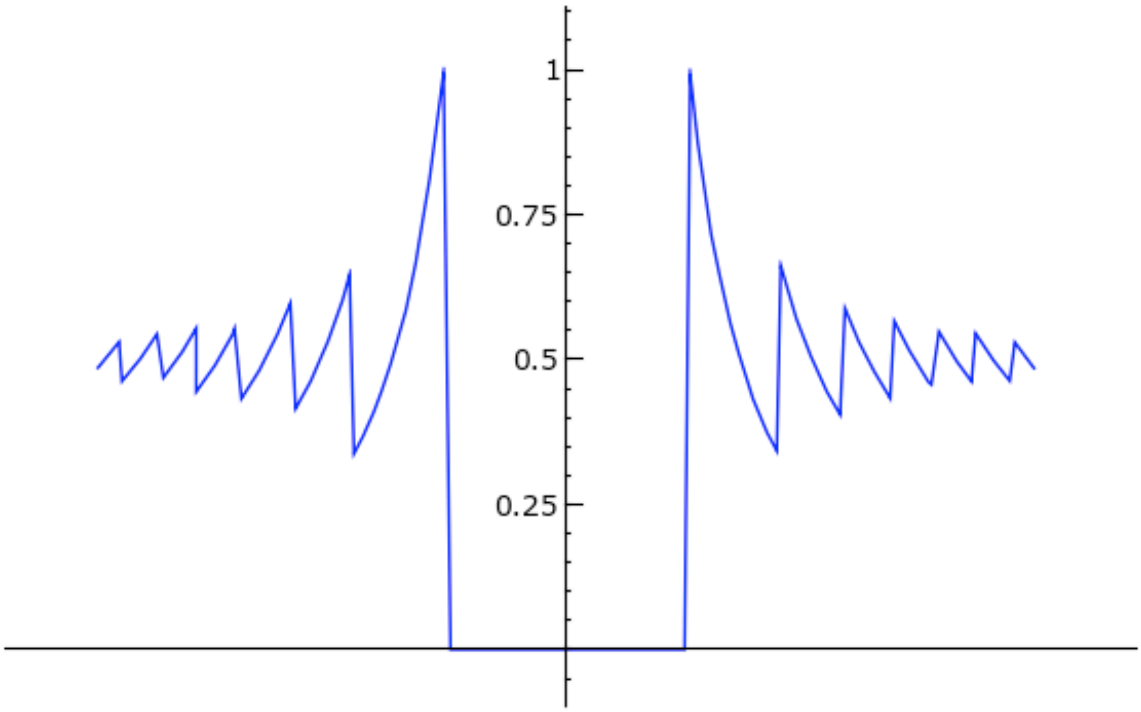
```
    [-4.5208202449859101e-16 .. 6.5814107953590434e-16]
```

```
a.contains_zero()
```

```
    True
```

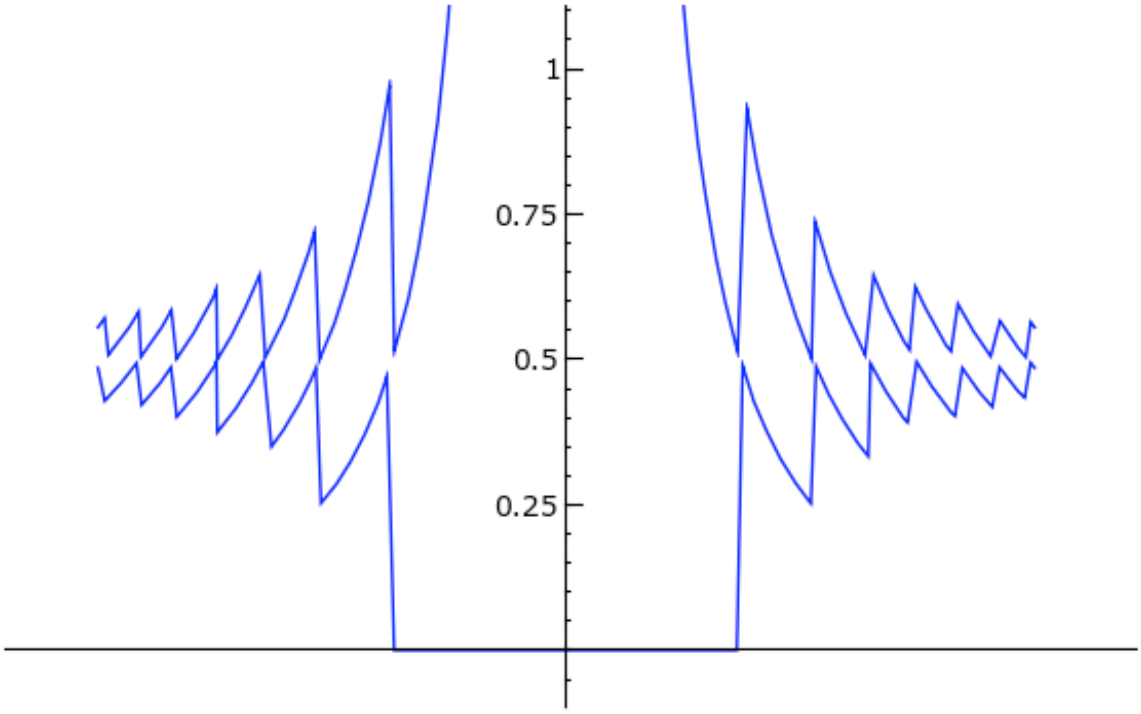
We create a plot but where with floats there is a lot of precision loss, which is very misleading.

```
var('x')
plot((1-cos(x))/x^2, (x,-4*10^(-8),4*10^(-8))).show(xmin=-4*10^(-
8),xmax=4*10^(-8), ymin=0)
```



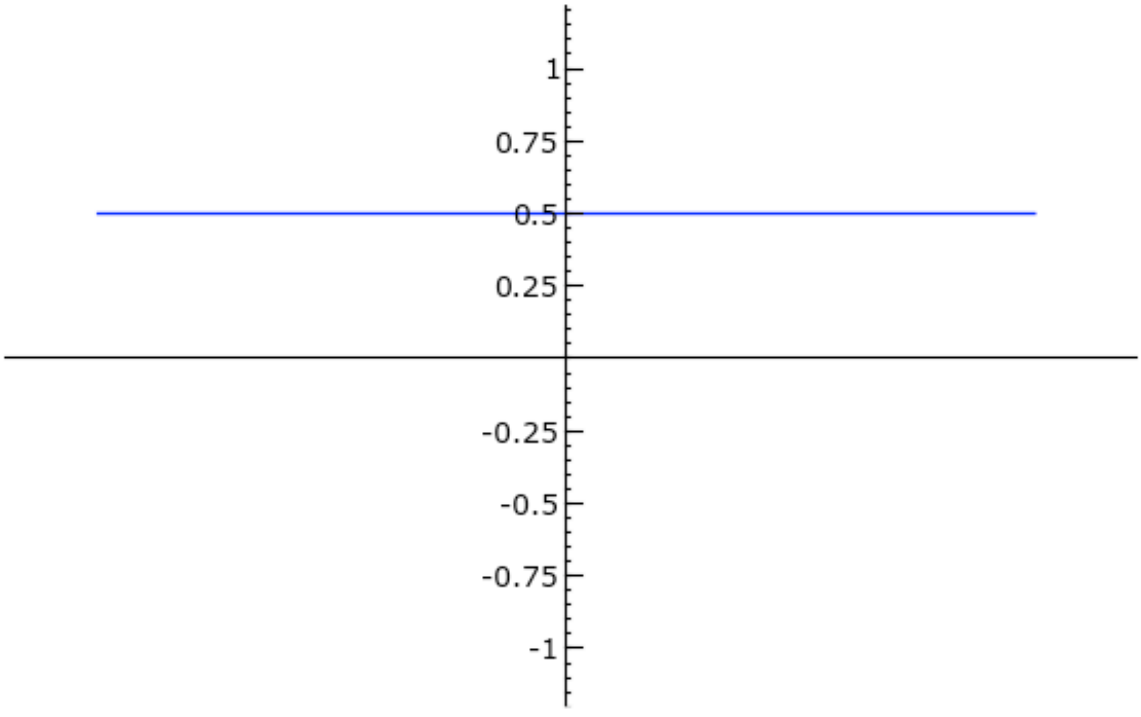
The same plot using intervals -- at least we see that something very funny might be going on:

```
def lower(x):
   y = RIF(x)
   return ((1-cos(y))/y^2).lower()
def upper(x):
   y = RIF(x)
   return ((1-cos(y))/y^2).upper()

plot((lower, upper), (x,-4*10^(-8),4*10^(-8))).show(xmin=-4*10^(-
8),xmax=4*10^(-8),ymax=1, ymin=0)
```



The same plot to much higher precision. This is what it should really look like.

```
def f(x):
   y = RealField(10000)(x)
   return ((1-cos(y))/y^2)

plot(f, (x,-4*10^(-8),4*10^(-8))).show(xmin=-4*10^(-8),xmax=4*10^(-8),ymax=1)
```



```
a = random_matrix(RIF,3); a
```

```
[[-2.0000000000000000 .. -2.0000000000000000] [-1.0000000000000000
.. -1.0000000000000000] [-2.0000000000000000 ..
-2.0000000000000000]]
[[-2.0000000000000000 .. -2.0000000000000000] [0.00000000000000000
.. 0.00000000000000000] [-2.0000000000000000 ..
-2.0000000000000000]]
[[-1.0000000000000000 .. -1.0000000000000000] [-2.0000000000000000
.. -2.0000000000000000]   [1.0000000000000000 ..
1.0000000000000000]]
```

```
a^2
```

```
[  [8.0000000000000000 .. 8.0000000000000000]   [6.0000000000000000
.. 6.0000000000000000]   [4.0000000000000000 .. 4.0000000000000000]]
[  [6.0000000000000000 .. 6.0000000000000000]   [6.0000000000000000
.. 6.0000000000000000]   [2.0000000000000000 .. 2.0000000000000000]]
[  [5.0000000000000000 .. 5.0000000000000000] [-1.0000000000000000
.. -1.0000000000000000]   [7.0000000000000000 ..
7.0000000000000000]]
```

```
a = matrix(RIF, 3, 3, [1/2, 2/3, 5, 7/8, -pi, e, e, sqrt(2), e])
```

```
a
```

```
[[0.50000000000000000 .. 0.50000000000000000] [0.66666666666666662
.. 0.66666666666666675]   [5.0000000000000000 ..
5.0000000000000000]]
[[0.87500000000000000 .. 0.87500000000000000] [-3.1415926535897936
.. -3.1415926535897931]   [2.7182818284590450 ..
2.7182818284590456]]
[  [2.7182818284590450 .. 2.7182818284590456]   [1.4142135623730949
.. 1.4142135623730952]   [2.7182818284590450 .. 2.7182818284590456]]
```

```
a^2
```

```
[[14.424742475628557 .. 14.424742475628563] [5.3100060428056110 ..
5.3100060428056147] [17.903597027934587 .. 17.903597027934595]]
[[5.0776625270395792 .. 5.0776625270395837] [14.297168762581803 ..
14.297168762581814] [3.2243218762570791 .. 3.2243218762570863]]
[[9.9856338802366284 .. 9.9856338802366338] [1.2135359756401111 ..
1.2135359756401170] [24.824696269384986 .. 24.824696269385001]]
```

```
a.charpoly()
```

```
[1.0000000000000000 .. 1.0000000000000000]*x^3 +
[-0.076689174869259525 .. -0.076689174869243537]*x^2 +
[-26.770363139026746 .. -26.770363139026496]*x +
[-46.034245822686686 .. -46.034245822686230]
```

```
restore('RealNumber')
a = float(1.2939393)
```

```
time for _ in xrange(10^6): b = a*a
```

```
    CPU time: 0.25 s,  Wall time: 0.26 s
```

```
a = RDF(1.2939393)
```

```
time for _ in xrange(10^6): b = a*a
```

```
    CPU time: 0.25 s,  Wall time: 0.25 s
```

```
restore('RealNumber')
a = RIF(pi,pi+0.0001)
```

```
time for _ in xrange(10^6): b = a*a
```

```
    CPU time: 0.89 s,  Wall time: 0.90 s
```

```
0.7 / 0.15
```

```
    4.66666666666667
```

```
sage: u, v = var('u,v')
       sage: fx = v *cos(u) - 0.5* v^2 * cos(2* u)
       sage: fy = -v *sin(u) - 0.5* v^2 * sin(2* u)
       sage: fz = 4 *v^1.5 * cos(3 *u / 2) / 3
       sage: parametric_plot3d([fx, fy, fz], (u, -2*pi, 2*pi), (v,
0, 1),plot_points = [90,90], frame=False,color="purple",
opacity=0.5) + sphere((0,0,0),1/2,aspect_ratio=[1,1,1])
```

```
jmol
```