480 -- 05-07-2008 floating point

Floating Point Arithmetic in Sage

A reference: What Every Computer Scientist Should Know About Floating-Point Arithmetic, by David Goldberg (see http://docs.sun.com/source/806-3568/ncg_goldberg.html).

Sage has a Plethora of Different Floating Point Arithmetic Models

Python Types: float, complex, decimal
Native Sage Types: RDF, CDF, RQDF, CC, RR, RIF, CIF
Types in Systems Sage Includes: pari, maxima

Python Types

float: Python double-precision floats

From the Python docs: "Floating point numbers are implemented using double in C. *All bets on their precision* are off unless you happen to know the machine you are working with."

```
float?
```
```
    <type 'float'>
```
```
a = float('12.93939'); a
```
```
    12.93939
```
```
time for _ in xrange(10^6): b = a*a
```
```
    CPU time: 0.25 s,  Wall time: 0.25 s
```
```
a^100
```

```
    1.5537469625197027e+111
```

```
a^300
```

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

```
1/float(0)
```

```
    Traceback (click to the left for traceback)
    ...
    ZeroDivisionError: float division
```

complex: Python double-precision complex

Much like Python floats. *All bets on their precision* are off unless you happen to know the machine you are working with.

```
complex?
```

```
a = complex(-2.393,3.3049)
```

```
a
```

```
    (-2.3929999999999998+3.3048999999999999j)
```

```
a.real
```

```
    -2.3929999999999998
```

```
a.imag
```

```
    3.3048999999999999
```

```
a.conjugate()
```

```
    (-2.3929999999999998-3.3048999999999999j)
```

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

```
    CPU time: 0.26 s,  Wall time: 0.28 s
```

decimal: Python arbitrary precision floats

Decimal floating point has finite precision with arbitrarily large bounds. From the Python docs: "The purpose of this module is to support arithmetic using familiar "schoolhouse" rules and to avoid some of the tricky representation issues associated with binary floating point. The package is especially useful for financial applications or for contexts where users have expectations that are at odds with binary floating point (for instance, in binary floating point, 1.00 % 0.1 gives 0.09999999999999995 instead of the expected Decimal("0.00") returned by decimal floating point)." Here are some examples of using the decimal module:

```
from decimal import *
setcontext(ExtendedContext)
Decimal(0)
```

```
    Traceback (click to the left for traceback)
    ...
    TypeError: Cannot convert 0 to Decimal
```

```
Decimal(int(0))
```

```
    Decimal("0")
```

```
Decimal("1")
```

```
    Decimal("1")
```

```
Decimal("-.0123")
```

```
    Decimal("-0.0123")
```

```
Decimal('123456')
```

```
    Decimal("123456")
```

```
Decimal("123.45e12345678901234567890")
```

```
    Decimal("1.2345E+12345678901234567892")
```

```
Decimal("1.33") + Decimal("1.27")
```

```
    Decimal("2.60")
```

```
Decimal("12.34") + Decimal("3.87") - Decimal("18.41")
```

```
    Decimal("-2.20")
```

```
dig = Decimal('1')
dig / Decimal('3')
```

```
    Decimal("0.333333333")
```

```
C = getcontext()
```

```
C.prec = 18
dig / Decimal('3')
```

```
    Decimal("0.333333333333333333")
```

I really do *NOT* like the Python builtin decimal module, at least for real mathematics.

1. It is *vastly slower* even than Sage's RealField.
2. The output printing format is ugly.
3. It uses global contexts to determine how arithmetic works instead of the idea of a parent field, which is an extremely difficult and backward programming model for arithmetic.
4. It looks, feels, and works differently than the other Python numeric types.

```
from decimal import *
setcontext(ExtendedContext)
a = Decimal('9.23903840923')
```

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

```
    CPU time: 32.51 s,  Wall time: 33.83 s
```

```
a*a
```

```
    Decimal("85.3598307")
```

```
aa = RR(a)
aa*aa
```

```
    85.3598307272272
```

```
time for i in xrange(10^6): b = aa*aa
```

```
    CPU time: 0.60 s,  Wall time: 0.61 s
```

Native Sage Types

RDF and CDF: Double Precision Sage Types

RDF and CDF are Sage's double precision real and complex number types. Both are built on top of GSL types, where GSL is the Gnu scientific library. They provide both arithmetic and lots of special functions. The main reasons to use RDF and CDF in Sage are (1) speed, (2) they fit well into the usual Sage structure of elements, parents, etc. In particular, they work much much better with the rest of Sage than Python floats. **WARNING:** Like with Python floats, *all bets on their precision* are off unless you happen to know the machine you are working with.

Also, **matrix algebra** with matrices that have RDF and CDF entries is very fast compared to RR and CC matrices.

```
RDF
```
```
    Real Double Field
```
```
CDF
```
```
    Complex Double Field
```
```
a = RDF(1.399); a
```
```
    1.399
```
```
time for _ in xrange(10^6): b = a*a
```
```
    CPU time: 0.25 s,  Wall time: 0.31 s
```
```
b = RDF.pi(); b
```
```
    3.14159265359
```
```
a*b
```
```
    4.39508812237
```

Let's compare the speed of RDF and Python floats (they are similar, though Python is still a little faster)

```
timeit('a*b')
```
```
    625 loops, best of 3: 147 ns per loop
```
```
aa=float(a); bb=float(b)
```
```
timeit('aa*bb')
```
```
    625 loops, best of 3: 133 ns per loop
```

```
timeit('a.sin()')
```

```
    625 loops, best of 3: 739 ns per loop
```

```
timeit('math.sin(aa)')
```

```
    625 loops, best of 3: 431 ns per loop
```

```
# More than just arithmetic!!
R.<x> = RDF[]
R
```

```
    Univariate Polynomial Ring in x over Real Double Field
```

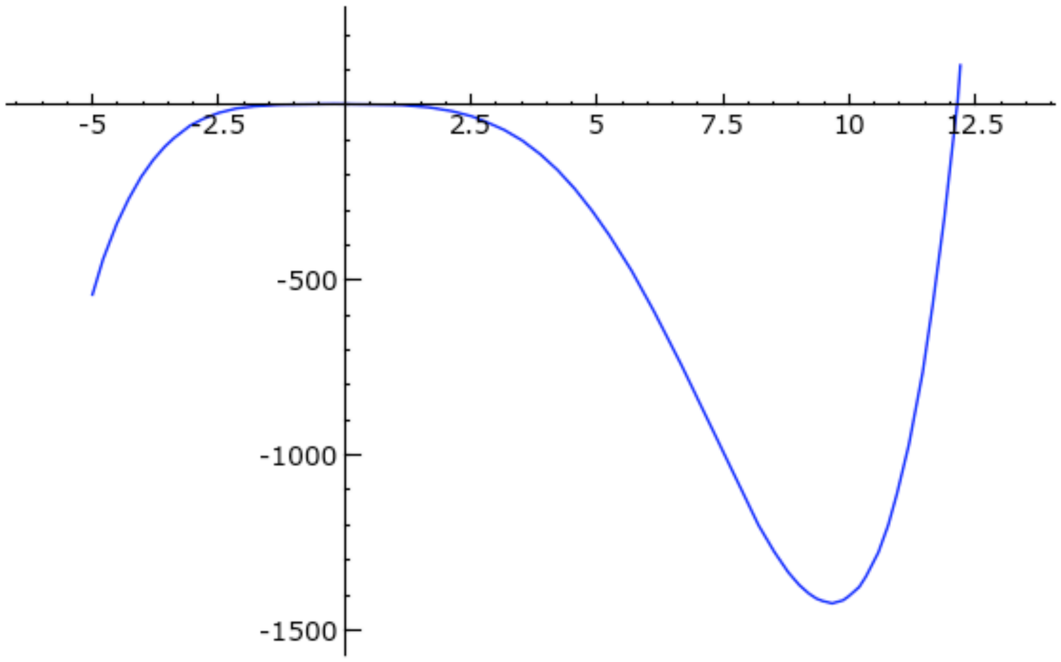
```
f = R.random_element(5); show(f)
```

$0.0602196534944x^{5} - 0.675014009648x^{4} - 0.625337010755x^{3} - 0.469194211969x^{2} - 0.252$

```
f.roots()
```

```
    [(-1.18657304832, 1), (0.708720861278, 1), (12.1207083604, 1)]
```

```
plot(f,-5,12.2)
```



```
r = random_matrix(RDF,4); show(r)
```

$$\left(\begin{array}{rrrr}
0.554793328163 & 0.806681249876 & 0.855672544804 & 0.345503002004 \\
0.275560503407 & -0.767516885245 & 0.140715689923 & -0.0672870712173 \\
0.411936035159 & -0.500777685361 & -0.25129942636 & -0.402880480964 \\
-0.0771603151762 & -0.102867639532 & 0.618852904735 & -0.924382081638
\end{array}\right)$$

```
show(r.charpoly())
```

$$1.0x^4 + 1.38840506508x^3 - 0.178664763023x^2 - 1.26500849771x - 0.559483646877$$

```
r.eigenspaces()
```

```
[(0.934869625022, Vector space of degree 4 and dimension 1 over
Complex Double Field
User basis matrix:
[   -0.78163648851   -0.195584588601   -0.592194261068
-0.00985016193641]), (-0.815028414943 + 0.446137143184*I, Vector
space of degree 4 and dimension 1 over Complex Double Field
User basis matrix:
[ 0.0954407885658 - 0.11522487684*I
-0.697830743559  0.185183691163 + 0.462464655174*I -0.475373712645
0.128451600015*I]), (-0.815028414943 - 0.446137143184*I, Vector
space of degree 4 and dimension 1 over Complex Double Field
User basis matrix:
[ 0.0954407885658 + 0.11522487684*I
-0.697830743559  0.185183691163 - 0.462464655174*I -0.475373712645
0.128451600015*I]), (-0.693217860214, Vector space of degree 4 and
dimension 1 over Complex Double Field
User basis matrix:
[0.0926455891765 -0.933148341712  0.316900596291 -0.142214553535])
```

Compare CDF and Python's complex type:

```
a = CDF(1,1); b = CDF(2,3.14)
aa = complex(a); bb = complex(b)
```

```
timeit('a*b')        # this seems too slow (?)
```

```
625 loops, best of 3: 282 ns per loop
```

```
timeit('aa*bb')
```

```
625 loops, best of 3: 178 ns per loop
```

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

```
CPU time: 0.39 s,  Wall time: 0.42 s
```

Note that RDF numbers can't be very big. This can and does cause trouble in everyday calculations, at least for people like me.

```
a = RDF(10)
```

```
a^308
```

```
    1e+308
```

```
a^309
```

```
    inf
```

RQDF: Quaddouble

The web page is here: http://crd.lbl.gov/~dhbailey/mpdist/

1. The idea of quaddouble is that four doubles in a row gives higher precision doubles.
2. It's a simple data structure created by computer scientists (mainly Hida at Berkeley). It is easier than mpfr to use on massively parallel computers and to extend the standard float linear algebra libraries to work with quaddouble.
3. It is nowhere near as rigorous as MPFR -- it gives all kinds of wrong answers in some cases in low order bits.
4. Overall it is not *that* much faster than MPFR. Special functions are about twice as fast.

```
RQDF
```

```
    Real Quad Double Field
```

```
a = RQDF.pi(); a
```

```
    3.141592653589793238462643383279502884197169399375105820974944590
```

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

```
    CPU time: 0.67 s,  Wall time: 0.70 s
```

```
a*a
```

```
    9.869604401089358618834490999876151135313699407240790626413349361
```

```
# compare with RR (mpfr)
a = RealField(212).pi(); a
```

```
    3.14159265358979323846264338327950288419716939937510582097494459
```

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

```
    CPU time: 0.75 s,  Wall time: 0.84 s
```

RR: Real Multiprecision (built on MPFR)

Paul Zimmerman is the author of MPFR. This following description is from the MPFR web page:
MPFR (see http://www.mpfr.org/) is a portable library written in C for arbitrary precision arithmetic on floating-point numbers. It is based on the GNU MP library. It aims to extend the class of floating-point numbers provided by the GNU MP library by a precise semantics. The main differences with the mpf class from GNU MP are:

- the mpfr code is portable, i.e. the result of any operation does not depend (or should not) on the machine word size mp_bits_per_limb (32 or 64 on most machines);
- the precision in bits can be set exactly to any valid value for each variable (including very small precision);
- mpfr provides the four rounding modes from the IEEE 754-1985 standard.

In particular, with a precision of 53 bits, mpfr should be able to exactly reproduce all computations with double-precision machine floating-point numbers (double type in C), except the default exponent range is much wider and subnormal numbers are not implemented but can be emulated.

Sage and Magma are the only general purpose computer algebra systems that use MPFR for their multiprecision floating point arithmetic. In Sage MPFR is used under the hood to implement RR and CC.

SUMMARY: Sage's RR and CC fields are **awesome** from a precision point of view because their precision behavior is rigorous and completely independent of the machine on which they are being run. Very very nice.

```
RR
```
```
    Real Field with 53 bits of precision
```
```
RealField(53) is RR
```
```
    True
```
```
RealField(200)
```
```
    Real Field with 200 bits of precision
```
```
a = RealField(53).pi()
```
```
time for _ in xrange(10^6): b = a*a
```
```
    CPU time: 0.59 s,  Wall time: 0.62 s
```
```
R = RealField(1000); R
```
```
    Real Field with 1000 bits of precision
```
```
R(pi)
```
```
    3.14159265358979323846264338327950288419716939937510582097494459230
    8164062862089986280348253421170679821480865132823066470938446095505
    2231725359408128481117450284102701938521105559644622948954930381964
    2881097566593344612847564823378678316527120190914564856692346034861
    454326648213393607260249141273
```
```
R(pi^2 + e - 1/sqrt(2))
```
```
    11.8807794483618563297939341091239645937861105632522761647919771348
    7552135416993181479618150798978728839853825905700554871273436907321
    6941888912246958253760219208518707663249568398958630171330016026288
    3963670673835883408960426120650299492612743293591820480688273056400
    4932925888819917512236470
```

CC: Complex Multiprecision

```
CC
```

```
    Complex Field with 53 bits of precision
```

```
ComplexField(200)
```

```
    Complex Field with 200 bits of precision
```

```
a = CC(pi + I)
```

```
time for _ in xrange(10^6): b = a*a      # surprisingly slow?.
```

```
    CPU time: 1.58 s,  Wall time: 1.61 s
```

```
a = ComplexField(300)(pi + 2*I); a
```

```
    3.1415926535897932384626433832795028841971693993751058209749445923
    81640628620899862803482 +
    2.0000000000000000000000000000000000000000000000000000000000000000
    000000000000000000000*I
```

```
a^3
```

```
    -6.692835162777698686075405532252639408140744226616162157554797003
    740195796880194596971688 +
    51.217626406536151713006945999256906811882196443444743758480096257
    02689345152314580106404*I
```

```
# only 8 bits of precision
a = ComplexField(8)(pi + 2*I); a
```

```
    3.1 + 2.0*I
```

```
a^3
```

```
    -6.6 + 51.*I
```

```
RIF
```

```
    Real Interval Field with 53 bits of precision
```

```
CIF
```

```
    Complex Interval Field with 53 bits of precision
```

RIF, CIF: Sage's Interval Arithmetic

More about this next time... This allows us to be **much** more confident in the output of a calculation at the expense of speed. Basically we won't get fooled by rounding errors.

```
RealIntervalField(200)(pi)
```

```
    [3.1415926535897932384626433832795028841971693993751058209749444 .
    3.1415926535897932384626433832795028841971693993751058209749470]
```

pari, maxima: Float types in Othe Software Sage Includes

PARI

```
PARI implements its own arbitrary precision real numbers.  This is
completely
separate from MPFR (and I think much older).  PARI has far more
high precision
special functions than any other free open source program in
existence.  Sage
uses PARI to implement many of the special functions on elements of
CC (but not RR).
```

```
    Syntax Error:
        PARI implements its own arbitrary precision real numbers.  Thi
    is completely
```

```
a = pari(pi)
a
```

```
    3.1415926535897932384626433833
```

```
a.gamma()
```

```
    2.2880377953400324179595889091
```

```
a^2
```

```
    9.8696044010893586188344910000
```

```
time for _ in xrange(10^6): b = a*a
```
```
    CPU time: 6.30 s,  Wall time: 6.59 s
```

```
# WARNING! The above sloness is not PARI being slow per se, but the
Sage interface to pari, as
# pari can do this calculation by itself quickly:
```

```
%gp
a = Pi;
gettime;
for(i=1,10^6,b=a*a);
gettime/1000.0
```
```
    0.4480000000000000000000000000
```

```
# With pari one sets a global precision that impacts all further
calculations.
pari.set_real_precision(200)
```
```
    28
```

```
a = pari(pi); a
```
```
    3.14159265358979323846264338327950288419716939937510582097494459230
    81640628620899862803482534211706798214808651328230664709384460955
    22317253594081284811174502841027019385211055596446229489549303820
```

```
a.gamma()
```
```
    2.2880377953400324179595889090602339228896881533562224411993807454
    4710066085042825007253044679284747968492456163619736990086693068611
    84720719929526723030053489815429191959102611884193668794490992030
```

```
# This converts MPFR pairs of numbers back and for to PARI behind
the scenes
# to do this calculation.
a = ComplexField(500)(pi + I)
a.zeta()
```
```
    1.0977390684282594480881320973507592738706227822476680574144542084
    4472957803611793715828949049670485542759065100807399292344514634 70
    851762576539018 -
    0.1287546531500526746334861685306011514356104721647399722702804527
    0436073469222422304441369858108450129450919873390290321044022824 41
    463916532960241 3*I
```

```

```

Maxima

```
maxima('1.399023840928340982309482903849023840982340 82')
```

```
    1.399023840928341
```

```
maxima.eval('fpprec : 100')
```

```
    '100'
```

```
maxima('1.399023840928340982309482903849023840982340 82')
```

```
    1.399023840928341
```

```
a = maxima(pi).bfloat()
a
```

```
    3.14159265358979323846264338327950288419716939937510582097494459 23
    816406286208998628034825342117068b0
```

```
a*a
```

```
    9.869604401089358618834490999876151135313699407240790626413349376 2
    044822419205243001773403718552231b0
```

```
time for _ in xrange(10^3): b = a *a
```

```
    CPU time: 3.24 s,  Wall time: 8.09 s
```

Binary Versus Decimal and Roundoff

From http://mathworld.wolfram.com/RoundoffError.html. The Patriot missile defense system used during the Gulf War was also rendered ineffective due to roundoff error (Skeel 1992, U.S. GAO 1992). The system used an integer timing register which was incremented at intervals of 0.1 s. However, the integers were converted to decimal numbers by multiplying by the binary approximation of 0.1,

$$0.00011001100110011001100_2 = \frac{209715}{2097152}.$$

As a result, after 100 hours (3.6×10^6 ticks), an error of

$$(\frac{1}{10} - \frac{209715}{2097152}) \cdot 3600000 = \frac{5625}{16384}$$

or approx. 0.3433 seconds had accumulated. This discrepancy caused the Patriot system to continuously recycle itself instead of targeting properly. As a result, an Iraqi Scud missile could not be targeted and was allowed to detonate on a barracks, killing 28 people.