71-APROX-ejemplos

February 5, 2018

Ejemplo -1

```
In [1]: NR0 = RealField(prec=2048)

In [2]: NR0(pi)

Out[2]: 3.14159265358979323846264338327950288419716939937510582097494459230781640628620899862
        25342117067982148086513282306647093844609550582231725359408128481117450284102701938521
        59644622948954930381964428810975665933446128475648233786783165271201909145648566923460
        10454326648213393607260249141273724587006606315588174881520920962829254091715364367892
        60011330530548820466521384146951941511609433057270365759591953092186117381932611793105
        48074462379962749567351885752724891227938183011949129833673362440656643086021394946395
        37190702179860943702770539217176293176752384674818467669405132000568127145264

In [3]: C = str(NR0(pi))

In [4]: len(C)

Out[4]: 617

In [5]: C[:2]

Out[5]: '3.'

In [6]: (2048/(617-1)).n() #bits por cifra en promedio

Out[6]: 3.32467532467532

In [7]: N(pi,digits=616)

Out[7]: 3.14159265358979323846264338327950288419716939937510582097494459230781640628620899862
        25342117067982148086513282306647093844609550582231725359408128481117450284102701938521
        59644622948954930381964428810975665933446128475648233786783165271201909145648566923460
        10454326648213393607260249141273724587006606315588174881520920962829254091715364367892
        60011330530548820466521384146951941511609433057270365759591953092186117381932611793105
        48074462379962749567351885752724891227938183011949129833673362440656643086021394946395
        37190702179860943702770539217176293176752384674818467669405132000568127145264
```

Ejemplo 0

```
In [8]: 1.1+1.1+1.1==3.3 #No son iguales £Por qué?

Out[8]: False

In [9]: bool(1.1+1.1+1.1==3.3+0.0)

Out[9]: False

In [10]: type(1.1+1.1+1.1)==type(3.3+0.0)

Out[10]: True

In [11]: 1.1+1.1+1.1-3.3

Out[11]: 4.44089209850063e-16
```

Debido a los errores de redondeo, $1.1+1.1+1.1-3.3 $ resulta ser, para el ordenador, del orden de $4 \cdot 10^{-16}$ y, por tanto no es cero. Cuando preguntamos a SAGE si dos enteros o racionales son iguales la respuesta es fiable, si en cambio son decimales NO.

```
In [1]: R = RealField(2048)

In [2]: R(1.1+1.1+1.1-3.3)

Out[2]: 4.44089209850062616169452667236328125000000000000000000000000000000000000000000000000000000000000000000000000000000000000
```

Aunque aumentemos la precisión se obtiene la misma respuesta. Sin embargo:

```
In [4]: R(1.1+1.1+1.1+1.1-4.4);print R(1.1+1.1+1.1+1.1)==R(4.4);1.1+1.1+1.1+1.1==4.4

False


Out[4]: True

In [5]:  R(1.1+1.1+1.1+1.1)-R(4.4)

Out[5]: 3.552713678800500929355621337890624999999999999999999999999999999999999999999999999999999999999999999999999999999999999
```

```
In [6]: 1.1+1.1+1.1+1.1-4.4

Out[6]: 0.000000000000000
```

En este otro cálculo los errores de redondeo no afectan al resultado y la respuesta es "correcta". £A qué se puede deber que primero responde *False* y después *True*?

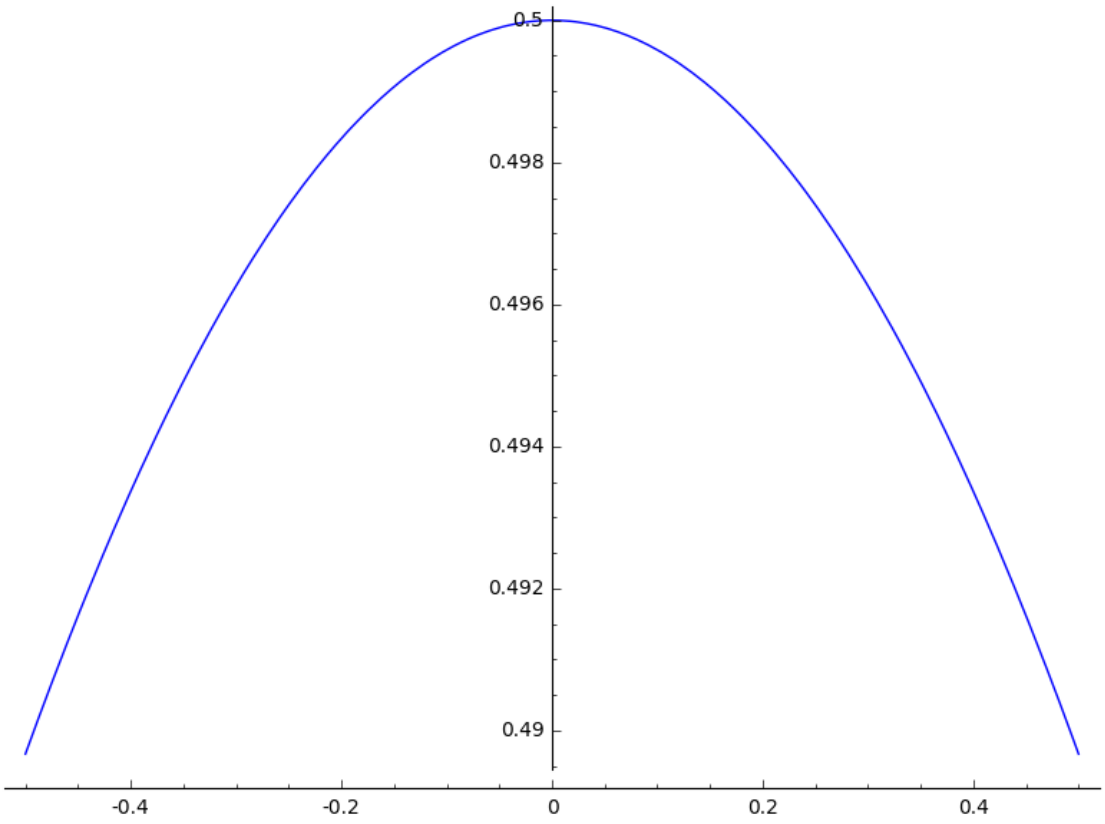
Ejemplo 1

```
In [7]: NR = RealField(prec=30)

In [8]: f(x)=(1-cos(x))/x^2

In [9]: plot(f,-0.5,0.5)
```

Out[9]:




```
In [10]: taylor(f,x,0,5)
```

```
Out[10]: x |--> 1/720*x^4 - 1/24*x^2 + 1/2
```

```
In [11]: g(x) = cos(x)
```

```
In [12]: g(x=NR(1.2*10^(-5)))
```

```
Out[12]: 1.0000000
```

Como, con $prec = 30$, el $cos(1.2 * 10^{-5})$ vale 1.0000000 el valor de $f(1.2 * 10^{-5})$ sale cero con un error enorme.

```
In [13]: V = f(x=NR(1.2*10^(-5)));V
```

```
Out[13]: -0.00000000
```

```
In [14]: NR1 = RealField(prec=40)
```

```
In [15]: f(x=NR1(1.2*10^(-5)))
```

```
Out[15]: 0.49895889889

In [24]: NR2 = RealField(prec=56)

In [25]: f(x=NR2(1.2*10^(-5)))

Out[25]: 0.5000000220954802

In [26]: NR3 = RealField(prec=1024)

In [27]: V1 = f(x=NR3(1.2*10^(-5)));V1

Out[27]: 0.49999999994000000000028799695989636780840194604989313727243425399934554887021416053
         73084561728572250243222378660379664772249431428110655012248641929113976288317718787062
         61918216625789151512122348687256530396835749357218404676015397109224469248967075310131
         16519612935761602332255464241487904564185436389731293772683431174717898230881894495412
```

Vemos que con $prec = 40$ son correctas dos de las cifras decimales, aumentando a $prec = 56$ se pierden las dos cifras correctas y pasándose con $prec = 1024$ se vuelve a obtener un resultado menor que $1/2$, no es difícil demostrar que $f(x) \leq 1/2$ para $x \neq 0$, que debería tener bastantes cifras decimales correctas. El error que se produce en V se llama de "cancelación" , y se define como un error enorme que aparece al restar dos cantidades muy próximas que sólo difieren en cifras más allá de la precisión que estamos usando en el cálculo.

```
In [28]: NR4 = RealField(prec=2048)

In [29]: V2 = f(x=NR4(1.2*10^(-5)));V2

Out[29]: 0.49999999994000000000028799695989636780840194604989313727243425399934554887021416053
         73084561728572250243222378660379664772249431428110655012248641929113976288317718787062
         61918216625789151512122348687256530396835749357218404676015397109224469248967075310131
         16519612935761602332255464241487045641854363897312937726834311747178982308818944954123
         08759274896856252209287844976930965263747215886764338260301368363634959716183264038333
         63029334619546079641686055403439399682035198133210422087357180897095371464404663686548
         35376715290751579347033783957331994215098859503968106881873977466511987813572 1
```

Al aumentar la precisión a $prec = 2048$ sólo las últimas 7 cifras decimales de $V1$ no se mantienen. El error que se produce en esas últimas 7 cifras de $V1$ es el inherente al cálculo con decimales aproximados ("error de redondeo") y es inevitable. Los errores de cancelación, en general, se pueden evitar aumentando la precisión pero los de redondeo son inevitables y debemos tratar de mantenerlos controlados.

Ejemplo 2

Otro ejemplo de cancelación, ahora en el cálculo de las raíces de una ecuación de segundo grado:

```
In [30]: f(x)=(10^(-20))*x^2-3*10^(20)*x+2*10^(20)

In [31]: sols = solve(f,x,solution_dict=True);sols
```

```
Out[31]: [{x: -10000000000000000000*sqrt(22499999999999999999999999999999999999998) +
         150000000000000000000000000000000000000}, {x:
         10000000000000000000*sqrt(22499999999999999999999999999999999999998) +
         150000000000000000000000000000000000000}]

In [32]: for sol in sols:
             print sol[x].n(digits=10)

Out[32]: 0.0000000000
         3.000000000e40

In [33]: for sol in sols:
             print sol[x].n(digits=20)

Out[33]: 0.00000000000000000000
         3.00000000000000000000e40

In [34]: for sol in sols:
             print sol[x].n(digits=40)

Out[34]: 0.6250000000000000000000000000000000000000
         3.0000000000000000000000000000000000000000e40

In [35]: for sol in sols:
             print sol[x].n(digits=50)

Out[35]: 0.66666666667151730507612228393554687500000000000000
         2.99999999999999999999999999999999999999333333333e40
```

Ejemplo 3

De la misma forma en que podemos obtener cancelaciones cuando una diferencia es muy próxima a 0, podemos obtenerlas cuando multiplicamos por cantidades muy próximas a 1, que la máquina toma como 1:

```
In [36]: def prueba(x,k):
             for muda in srange(k):
               x = sqrt(x)
             for muda in srange(k):
               x = x*x
             return x

In [37]: [prueba(100.0,10*k) for k in srange(1,8)]

Out[37]: [100.000000000006, 100.000000011555, 99.9999770096763, 99.9810245754440, 90.017126937
         1.00000000000000, 1.00000000000000]

In [38]: NR7 = RealField(prec=128)
```

```
In [39]: def prueba2(x,k):
             for muda in srange(k):
               x = NR7(sqrt(x))
             for muda in srange(k):
               x = NR7(x*x)
             return x

In [40]: [prueba2(100.0,10*k) for k in srange(1,8)]

Out[40]: [99.999999999999999999999999999999915, 100.000000000000000000000000000000360,
         99.999999999999999999999999984601618, 99.999999999999999999999700849351586,
         100.000000000000000000019874119658729, 100.000000000000000018416370944114861,
         99.999999999999996986404345071804247 62]
```

Vemos que también en este caso aumentando la precisión se resuelve el problema.

Ejemplo 4

En cálculos en Álgebra Lineal numérica (Álgebra Lineal con datos decimales) se encuentran con cierta frecuencia problemas de "inestabilidad": pequeños cambios en los datos producen cambios muy grandes en la solución calculada.

```
In [41]: def hilbert_m(n):
             A = matrix(RR,n,n,[0]*n^2)
             for fila in srange(n):
                 for columna in srange(n):
                     A[fila,columna]=(1/(fila+columna+1)).n()
             return A

In [42]: M = hilbert_m(5);M

Out[42]: [ 1.00000000000000 0.500000000000000 0.333333333333333 0.250000000000000
         0.200000000000000]
         [0.500000000000000 0.333333333333333 0.250000000000000 0.200000000000000
         0.166666666666667]
         [0.333333333333333 0.250000000000000 0.200000000000000 0.166666666666667
         0.142857142857143]
         [0.250000000000000 0.200000000000000 0.166666666666667 0.142857142857143
         0.125000000000000]
         [0.200000000000000 0.166666666666667 0.142857142857143 0.125000000000000
         0.111111111111111]

In [43]: b = M*vector([1,1,1,1,1]);b

Out[43]: (2.28333333333333, 1.45000000000000, 1.09285714285714, 0.884523809523809,
         0.745634920634921)

In [44]: M.solve_right(b)

Out[44]: (0.999999999999984, 1.00000000000015, 0.999999999999717, 1.00000000000002,
         1.00000000000013)
```

Resolvemos el sistema con matriz M y vector de términos independientes $b = M \cdot (1,1,1,1,1)^t$. La solución, como esperaríamos, es aproximadamente $(1,1,1,1,1)^t$. Perturbamos ligeramente el elemento $M[4,0]$ de M

```
In [45]: M[4,0]

Out[45]: 0.200000000000000

In [46]: M[4,0] += 10^(-5)

In [47]: M[4,0]

Out[47]: 0.200010000000000

In [48]: M.solve_right(b)

Out[48]: (0.993739441518408, 1.12521116963169, 0.436549736657753, 1.87647818742091,
          0.561760906289674)
```

Ahora la solución difiere mucho de $(1,1,1,1,1)^t$. Por supuesto, SAGE puede hacer este cálculo de manera exacta usando el cuerpo QQ de los números racionales en lugar de RR.