62-ARITM-fibonacci

December 3, 2017

Ejercicio 1

Cálculo de los números de Fibonacci usando matrices:

```
In [1]: A = matrix(ZZ,[[1,1],[1,0]])

In [2]: def potencia(A,n):
            k = A.ncols()
            if n == 0:
                return identity_matrix(k)
            elif n%2 == 0:
                B = potencia(A,n//2)
                return (B*B)
            else:
                B = potencia(A,(n-1)//2)
                return (A*B*B)

In [3]: AA = potencia(A,157)

In [4]: AA[0,0]

Out[4]: 468340976726457153752543329995929

In [5]: AA[0,0] == fibonacci(158)

Out[5]: True
```

Ejercicio 2

Hay fórmulas que permiten calcular el n-ésimo número de Fibonacci F_n sin haber calculado los anteriores. Una tal fórmula afirma que

$$F_n := \frac{\phi^n - (1-\phi)^n}{\sqrt{5}},$$

con ϕ el número áureo $\frac{1+\sqrt{5}}{2}$. OBSERVA que, aunque la parte derecha de la fórmula es una expresión complicada que involucra raíces cuadradas de 5, la fórmula está afirmando que el resultado es un número entero para todo valor de n, es decir, que las raíces se van a cancelar al operar.

En SAGE podemos operar de manera exacta (es decir, de forma "simbólica" como opuesta a "numérica") con expresiones como ϕ. Basta definir la raíz cuadrada de 5 como $sqrt(5)$ y usar

las operaciones aritméticas habituales. Se puede forzar a SAGE a desarrollar ("quitar los exponentes") una expresión como F_n usando la función *expand*. Comprueba que F_n es el número de Fibonacci n-ésimo para 20 valores de n mayores que 10000. Estudia la eficiencia de esta fórmula como medio para calcular el número de Fibonacci n-ésimo.

Define una función $fibonacci\_num(n, d)$ que devuelva el número F_n calculado, aproximadamente, como número real con d dígitos de precisión. Estudia la eficiencia de este método, comparando con el método del apartado 1 (siempre para valores de n mayores que 10000), y los errores que pueden aparecer al usar números reales con d dígitos de precisión.

```
In [4]: phi = (1+sqrt(5))/2
```

```
In [7]: def F(n):
            return expand(((phi^n)-(1-phi)^n)/sqrt(5))
```

```
In [10]: %time f = F(10000);
```

```
CPU times: user 6.48 s, sys: 42.5 ms, total: 6.52 s
Wall time: 6.32 s
```

```
In [ ]: del f
```

```
In [11]: %time ff = F(20000)
```

```
CPU times: user 16.3 s, sys: 160 ms, total: 16.4 s
Wall time: 16 s
```

```
In [13]: del ff
```

```
In [14]: all([F(n) == fibonacci(n) for n in xrange(10000,10020)])
```

```
Out[14]: True
```

Fibonacci usando números reales

```
In [1]: def fibonacci_num(n,d):
            phi2 = ((1+sqrt(5))/2).n(digits=d)
            return ((phi2^n-(1-phi2)^n)/(sqrt(5)).n(digits=d)).ceil()
```

```
In [2]: print [fibonacci_num(1000+k,200)-fibonacci(1000+k) for k in srange(1,30)]
```

```
[31497821971, 51081002696, 82780151259, 134129589411, 216909740670, 352113071905, 568485941663
```

```
In [3]: print [fibonacci_num(1000+k,300)-fibonacci(1000+k) for k in srange(1,100)]
```

```
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, (
```

Hace falta usar una precisión suficiente.

```
In [50]: print [fibonacci_num(10000,100*k)-fibonacci(10000) for k in srange(20,100)]

[-3134826036501491704504527040490793879607284446251030698713535186049022303931811493604497883

In [55]: print [fibonacci_num(10000,100*k)-fibonacci(10000) for k in srange(20,500)]

[-3134826036501491704504527040490793879607284446251030698713535186049022303931811493604497883
```

Vemos un desfase, que puede ser sólo de una unidad, pero que no desaparece al aumentar la precisión.

```
In [1]: def fibonacci_num2(n,d):
            phi2 = ((1+sqrt(5))/2).n(digits=d)
            return (phi2^n/sqrt(5).n(digits=d)).floor()

In [53]: print [fibonacci_num2(10000,100*k)-fibonacci(10000) for k in srange(20,100)]

[-3134826036501491704504527040490793879607284446251030698713535186049022303931811493604497883

In [54]: print [fibonacci_num2(10000,100*k)-fibonacci(10000) for k in srange(20,500)]

[-3134826036501491704504527040490793879607284446251030698713535186049022303931811493604497883
```

Parece que el resultado usando *floor* en lugar de *ceil* es mejor. Una tercera opción es usar *round* que devuelve el entero más próximo.

```
In [4]: def fibonacci_num3(n,d):
            phi2 = ((1+sqrt(5))/2).n(digits=d)
            return (phi2^n/sqrt(5).n(digits=d)).round()

In [5]: print [fibonacci_num3(10000,100*k)-fibonacci(10000) for k in srange(20,500)]

[-3134826036501491704504527040490793879607284446251030698713535186049022303931811493604497883
```

Usando *round* parece que una vez que se obtiene igualdad para una cierta precisión se mantiene para precisiones mayores.

```
In [6]: %time fibonacci_num3(100000,50000) == fibonacci(100000)

CPU times: user 412 ms, sys: 127 µs, total: 412 ms
Wall time: 413 ms


Out[6]: True
```

El cálculo simbólico es ciertamente muy lento, y "parece" que al aumentar el tamaño de los datos crece mucho el tiempo que tarda. En este tipo de problemas de cálculo algebraico exacto no es raro que el tiempo que tarda sea exponencial (o incluso doblemente exponencial e^{e^n}) como función del tamaño de los datos. NO hay motivo suficiente en los cálculos que he hecho para afirmarlo.

El método numérico es mucho más eficiente, pero hay que incrementar mucho el número de decimales en el cálculo para obtener una respuesta correcta. El problema es que no sabemos 'a priori' cuál es el d que debemos elegir como función de n, y aunque el método es bastante rápido, en cuanto incrementemos mucho el número de decimales inevitablemente se ralentiza.

```
In [21]: timeit('fibonacci_num(100000,40000)')

5 loops, best of 3: 245 ms per loop


In [22]: timeit('fibonacci_num(100000,100000)')

5 loops, best of 3: 945 ms per loop
```

Ejercicio 3

Dado un número de Fibonacci bastante grande F_n queremos calcular el lugar que ocupa en la sucesión, es decir, dado un F_n queremos obtener el n.

Define, usando la construcción iterativa de la sucesión de Fibonacci, una función que reciba como argumento F_n y devuelva n.

Cuando n crece, el segundo sumando (de hecho es un "restando") en el numerador del F_n del ejercicio anterior tiende a cero. Explica el motivo. Entonces, F_n es aproximadamente igual a $\phi^n/\sqrt{5}$, y esto podría servir para calcular n dado un F_n suficientemente grande. Implementa un tal método, y estudia su grado de validez comparando los valores de n que produce para variados F_n calculados a partir de enes dados.

```
In [5]: def subindice0(F):
            p,q=0,1
            cont = 0
            while p < F:
                p,q=q,p+q
                cont += 1
            return cont

In [6]: fibonacci(subindice0(fibonacci(10000))) == fibonacci(10000)

Out[6]: True

In [7]: (1-phi).n()

Out[7]: -0.618033988749895

In [8]: var('n');limit(((1-phi).n())^n,n=oo)

Out[8]: 0
```

```
In [9]: def subindice(F):
            return ((log(F*sqrt(5))/log(phi))).n().ceil()

In [10]: subindice(fibonacci(10000))

Out[10]: 10000

In [11]: [subindice(fibonacci(10^k))==10^k for k in srange(2,9)]

Out[11]: [True, True, True, True, True, True, True]
```

No parece que haya que aumentar la precisión para obtener resultados correctos para números de Fibonacci muy grandes. £Qué método es mejor?

```
In [13]: %time sub1 = subindice0(fibonacci(10^6))
         %time sub2 = subindice(fibonacci(10^6))

CPU times: user 4.87 s, sys: 54.6 ms, total: 4.93 s
Wall time: 4.85 s
CPU times: user 7.26 ms, sys: 0 ns, total: 7.26 ms
Wall time: 7.15 ms


In [14]: time sub2 = subindice(fibonacci(10^8))

CPU times: user 1.27 s, sys: 12 ms, total: 1.28 s
Wall time: 1.29 s
```

0.1 £Qué precisión nos conviene usar?

```
In [6]: def precision(N1,N2,d):
            cont = 0
            L = []
            for n in xsrange(N1,N2,7):
                while cont < 100:
                    cont += 1
                    if fibonacci_num3(n,d) - fibonacci(n) != 0:
                        cont = 0
                        d += 100
                L.append((n,d))
            return L

In [7]: %time precision(10000,10100,1000)

CPU times: user 344 ms, sys: 12 ms, total: 356 ms
Wall time: 347 ms
```

```
Out[7]: [(10000, 2100),
         (10007, 2100),
         (10014, 2100),
         (10021, 2100),
         (10028, 2100),
         (10035, 2100),
         (10042, 2100),
         (10049, 2100),
         (10056, 2100),
         (10063, 2100),
         (10070, 2100),
         (10077, 2100),
         (10084, 2100),
         (10091, 2100),
         (10098, 2100)]

In [8]: %time precision(100000,100200,1000)

CPU times: user 12.2 s, sys: 0 ns, total: 12.2 s
Wall time: 12.2 s


Out[8]: [(100000, 21000),
         (100007, 21000),
         (100014, 21000),
         (100021, 21000),
         (100028, 21000),
         (100035, 21000),
         (100042, 21000),
         (100049, 21000),
         (100056, 21000),
         (100063, 21000),
         (100070, 21000),
         (100077, 21000),
         (100084, 21000),
         (100091, 21000),
         (100098, 21000),
         (100105, 21000),
         (100112, 21000),
         (100119, 21000),
         (100126, 21000),
         (100133, 21000),
         (100140, 21000),
         (100147, 21000),
         (100154, 21000),
         (100161, 21000),
         (100168, 21000),
         (100175, 21000),
```

```
         (100182, 21000),
         (100189, 21000),
         (100196, 21000)]

In [8]: %time precision(112000,112200,1000)

CPU times: user 30.8 s, sys: 56 ms, total: 30.8 s
Wall time: 30.8 s


Out[8]: [(112000, 23500),
         (112007, 23500),
         (112014, 23500),
         (112021, 23500),
         (112028, 23500),
         (112035, 23500),
         (112042, 23500),
         (112049, 23500),
         (112056, 23500),
         (112063, 23500),
         (112070, 23500),
         (112077, 23500),
         (112084, 23500),
         (112091, 23500),
         (112098, 23500),
         (112105, 23500),
         (112112, 23500),
         (112119, 23500),
         (112126, 23500),
         (112133, 23500),
         (112140, 23500),
         (112147, 23500),
         (112154, 23500),
         (112161, 23500),
         (112168, 23500),
         (112175, 23500),
         (112182, 23500),
         (112189, 23500),
         (112196, 23500)]

In [10]: fibonacci_num3(112108,112108//4)-fibonacci(112108)

Out[10]: 0
```

Parece posible encontrar cómo depende la precisión necesaria del tamaño de los enteros que usamos como índices (el n tal que calculamos F_n). Con estas dos catas parece que la dependencia puede ser del orden $d = n/5$. Para terminar el ejercicio hay que plantear un experimento con un rango más significativo de valores de n, y usar gráficas y la función de Sage $find\_fit$ para obtener conclusiones. Observa que la función *precision* tiene parámetros fijos dentro del código

que pueden no ser óptimos, y, por tanto, quizá conviene convertirlos en parámetros de la función y experimentar también con ellos.

```
In [10]: %time all([fibonacci_num3(k,k//4)==fibonacci(k) for k in xsrange(5000,15000)])

CPU times: user 29.6 s, sys: 236 ms, total: 29.8 s
Wall time: 29.4 s


Out[10]: True
```

Usando $d = k/5$ no funcionó siempre bien, pero con $d = k/4$ vemos que, en este rango, siempre se obtiene que las dos funciones dan el mismo resultado.

```
In [12]: nucl = 16

         def fibonacci_num3(n,d):
             phi2 = ((1+sqrt(5))/2).n(digits=d)
             return ((phi2^n/sqrt(5)).n(digits=d)).round()



         def generar_lista(N1,N2,k,nucl):
             L = [[] for muda in range(nucl)]
             for n in xsrange(N1,N2,k):
                 r = n%nucl
                 if L[r] == []:
                     L[r]=[n]
                 else:
                     L[r] += [n]
             return L


         @parallel(nucl)
         def comprobar(L):
             L1 = []
             for n in L:
                 if fibonacci_num3(n,n//4) - fibonacci(n) != 0:
                         L1.append(n)
             return L1

         L = generar_lista(10**5,10**6,next_prime(1000),nucl)
         %time LL = list(comprobar([L[j] for j in srange(nucl)]))

CPU times: user 8 ms, sys: 84 ms, total: 92 ms
Wall time: 2min 20s


In [17]: print LL
```

```
[((([112108, 128252, 144396, 160540, 176684, 192828, 208972, 225116, 241260, 257404, 273548, 28
```

```
In [18]: print LL[0]
```

```
(((([112108, 128252, 144396, 160540, 176684, 192828, 208972, 225116, 241260, 257404, 273548, 289
```

```
In [19]: print LL[0][1]
```

```
[]
```

```
In [20]: for int in srange(16):
             print LL[int][1]
```

```
[]
[]
[]
[]
[]
[]
[]
[]
[]
[]
[]
[]
[]
[]
[]
[]
```

```
In [24]: print LL[0][0][0][0] in L
```

```
True
```

```
In [26]: for int in srange(16):
             print LL[int][0][0][0] in L
```

```
True
True
True
True
True
True
True
```

```
True
True
True
True
True
True
True
True
True
```

Parece que la precisión óptima estaría entre $d/5$ y $d/4$. Se podría afinar más porque *probablemente* algo más de $d/5$ también serviría.

```
In [ ]:
```