nombre.apellido-labodt-2017-ex2-SOL

February 12, 2018

1) Antes que nada, lee cuidadosamente estas instrucciones y las que aparecen en la hoja con la contraseña. .

2) Cambia el nombre de este archivo sustituyendo "nombre.apellido" por los tuyos, tal como aparecen en tu dirección de correo electrónico de la UAM.

3) Este archivo debe quedar en la carpeta "ENTREGA..." que está en el escritorio de tu cuenta de examen. Lo mejor es que esté en esa carpeta desde el comienzo del examen.

4) El examen resuelto debe quedar en este único archivo. No se puede usar un archivo para cada pregunta.

5) Recuerda que hay que deshabilitar el salvapantallas al comenzar el examen, tal como está indicado en la hoja con la contraseña.

CALIFICACIÓN:

In [ ]:

COMENTARIOS:

1)

0.1 Ejercicio 1

Podemos estimar el valor de $\alpha(x) := \sqrt{1+x}$ como $\beta(x) := 1 + (x/2)$ cuando x es, en valor absoluto, suficientement próximo a cero. Consideramos que la estimación *es aceptable* cuando el error relativo

$$E(x) := \frac{abs(\alpha(x) - \beta(x))}{\alpha(x)}$$

que cometemos es menor que 0.01.

1) Determina, experimentalmente, una cota $|x| < A$ tal que la estimación es aceptable para los x que la cumplen, y deja de serlo para $|x| \geq A$.

2) £Qué podemos hacer para mejorar la estimación? Postula una nueva estimación $\beta_1(x)$ y demuestra que con ella se obtienen resultados bastante más precisos.

```
In [1]: def aprox(x,precis):
            NR = RealField(prec=precis)
            return NR(1+(x/2))
```

```
In [2]: def raiz(x,precis):
            NR = RealField(prec=precis)
            return(NR(sqrt(1+x)))

In [3]: def error(x,y):
            return abs(x-y)/x

In [13]: print [(k,error(raiz(0.01*k,53),aprox(0.01*k,53))) for k in srange(-100,100) if error

[(-24, 0.00942922903047183), (-23, 0.00855110166779582), (-22, 0.00772656038869023), (-21, 0.00
```

La precisión no afecta al resultado, no era necesario usar precisión arbitraria, y esto parece indicar que el A buscado es del orden de $0.01 * 24 = 0.24$. Se podrían obtener resultados más finos usando 0.001 en lugar de 0.01 comop separación entre los x probados.

Para mejorar la estimación basta observar que $\beta(x)$ es el desarrollo de Taylor de primer orden en $x = 0$ de $\alpha(x)$, y entonces pasar al segundo orden.

```
In [14]: taylor(sqrt(1+x),x,0,2)

Out[14]: -1/8*x^2 + 1/2*x + 1

In [15]: def aprox2(x,precis):
             NR = RealField(prec=precis)
             return NR(1+(x/2)-(x**2/8))

In [16]: print [(k,error(raiz(0.01*k,53),aprox2(0.01*k,53))) for k in srange(-100,100) if erro

[(-44, 0.00998023318705171), (-43, 0.00914464619339355), (-42, 0.00836775114623932), (-41, 0.00
```

0.2 Ejercicio 2

John Napier publicó su invención de los logaritmos en 1614, mucho antes de la invención del cálculo diferencial. El punto esencial para que los logaritmos fueran útiles era la existencia de *tablas de logaritmos* en las que uno encontraba los logaritmos de los factores, que sumaba a mano, y volvía a usar para encontrar el número cuyo logaritmo era la suma obtenida, es decir, el producto de los números de partida. Henry Briggs colaboró con Napier para producir las primeras tablas de logaritmos mediante el siguiente procedimiento:

1) Queremos calcular el logaritmo de un entero $n > 1$. Calculamos $n^{1/2^K}$ para $K = 1, 2, 3, \ldots$ hasta que el resultado difiera *muy poco* de 1. Esto es lo mismo que iterar la extracción de raíces cuadradas hasta llegar casi a 1. Escribimos

$$n^{1/2^K} = 1 + x.$$

2) Ahora tomamos logaritmos para obtener

$$log(n) = 2^K log(1 + x).$$

3) Finalmente, Briggs observó que para x suficientemente pequeño en valor absoluto, podía sustituir $log(1+x)$ por x, y quedaba $log(n)$ aproximadamente igual a $2^K x$.

Es claro que lo que necesitamos es saber *cómo de pequeño tiene que ser x para obtener un número prefijado k de cifras decimales exactas del logaritmo de n.*

A) Define una función $buscar(n, k, precis)$ que debe efectuar el procedimiento de Briggs hasta que la estimación obtenida tenga el número k de cifras correctas de $log(n)$, y en ese momento debe devolver x. La precisión arbitraria en los cálculos hace falta porque sin ella un cierto bucle *while* puede hacerse infinito cuando n o k son *grandes*.

B) Experimenta con la función del primer apartado y, como consecuencia, enuncia y comprueba la regla que indica cómo de pequeño tiene que ser x para obtener k cifras correctas del logaritmo de n.

```
In [26]: def buscar(n,k,precis):
             NR = RealField(prec=precis)
             x = NR(log(n))
             y = NR(sqrt(n))
             K = 1
             while abs(x-2^K*(y-1))>10^(-k-1):
                 y = NR(sqrt(y))
                 K +=1
             return y-1,x,2^K*(y-1)
```

```
In [32]: buscar(23,3,2048)
```

```
Out[32]: (0.000047844989382549533055728786588895418998100430736819042053330135483044119752238
          3.135494215929149690806752831810196118442380314840435741998635377482993245984798291
          3.135569224174766198340241757889850179459509828768172740007043759016779432082725908
```

Parece que para obtener k cifras decimales correctas hay que imponer que x sea menor que 10^{-k-1}, ya que si sólo se impone menor que 10^{-k} en ocasiones sólo se obtienen $k-1$ cifras decimales correctas.

```
In [33]: def estimacion(n,k,precis):
             NR = RealField(prec=precis)
             y = NR(sqrt(n))
             K= 1
             x = y-1
             while x>10^(-k-1):
                 y= NR(sqrt(y))
                 x = y-1
                 K += 1
             return 2^K*x
```

```
In [34]: estimacion(23,3,2048)
```

```
Out[34]: 3.135644234812885643722946323526275964123895345794215994649002701737308812465729822352
```

```
In [35]: estimacion(23,10,2048)
```

```
Out[35]: 3.135494215938091227876287805612887966005780073329466532528538449956483685400361011000
```

0.3 Ejercicio 3

El método BBP que permite calcular una cifra determinada de π, la n-ésima, sin calcular las anteriores se puede aplicar a algunas otras constantes. Por ejemplo, para $log(2)$ se puede usar la serie

$$log(2) = \sum_{k=1}^{\infty} \frac{1}{2^k k}.$$

1) En la primera celda se reproduce el código BPP para π, y lo primero que debes hacer es modificarlo para que funcione para $log(2)$ mediante la serie indicada. La última función, la que devuelve cifras binarias de $log(2)$, debe llamarse $cifra\_log2(n)$.

2) Modifica el código del apartado 1) para haga todos los cálculos con una precisión (en bits) arbitraria, es decir, la nueva función debe llamarse $cifra\_log2(n, precis)$.

3) Define una función $compara(C1, C2)$ que, dadas dos cadenas de caracteres, cuente, hasta la primera discrepancia, el número de caracteres de $C1$ que ocupan el mismo lugar en $C2$. Es decir, si el primer caracter es diferente debe devolver cero, si el primero es igual pero el segundo es diferente debe devolver uno, etc.

4) Jugando con la función $cifra\_log2(n, precis)$ se observa que cuanto mayor es la precisión más cifras correctas de $log(2)$, a partir de la n-ésima, devuelve, y que fijada la precisión el número de cifras correctas no depende mucho de n y es del orden de la precisión. La parte final del ejercicio consiste en comprobar sistemáticamente estas afirmaciones, para lo que debemos comparar cadenas $C1$ generadas por $cifra\_log2(n, precis)$ con la parte relevante de cadenas $C2$

$$(log(2).n(prec = precis)).str(base = 2, no\_sci = 2)$$

que nos dan las cifras correctas de $log(2)$. El parámetro $no\_sci = 2$ sirve para que no se devuelva el resultado en notación científica estándar (i.e. parte entera de un único dígito, decimales, y exponente de una potencia de 10).

In [ ]:
```
####Codigo para las cifras de pi

    def F0(j,n):
        S =RR(0.0)
        k =0
        while k <= n:
            r = 8*k+j
            S += RR(power_mod(16,n-k,r)/r)-floor(RR(power_mod(16,n-k,r)/r))
            k += 1
        return RR(S)

    def F1(j,n):
        S =RR(0.0)
        k =n+1
        while 1:
            r = 8*k+j
            nS = S+ RR(16^(n-k)/r)
            if S == nS:
```

```
                break
            else:
                S = nS
            k += 1
        return RR(S)

    def S(j,n):
        return RR(F0(j,n)+F1(j,n))

    def cifra_pi(n):
        n -= 1
        x = (4*RR(S(1,n))-2*RR(S(4,n))-RR(S(5,n))-RR(S(6,n)))
        return (x-floor(x)).str(base=16,no_sci=2)
```

In [39]:
```
def F0(n,precis):
    NR = RealField(prec=precis)
    S =NR(0.0)
    k = 1
    while k <= n:
        S += NR(power_mod(2,n-k,k)/k)-floor(NR(power_mod(2,n-k,k)/k))
        k += 1
    return NR(S)

def F1(n,precis):
    NR = RealField(prec=precis)
    S =NR(0.0)
    k =n+1
    while 1:
        nS = S+ NR(2^(n-k)/k)
        if S == nS:
            break
        else:
            S = nS
        k += 1
    return NR(S)

def S(n,precis):
    NR = RealField(prec=precis)
    return NR(F0(n,precis)+F1(n,precis))

def cifra_log2(n,precis):
    NR = RealField(prec=precis,sci_not=False)
    n -= 1
    x = NR(S(n,precis))
    return (x-(floor(x))).str(base=2,no_sci=2)
```

In [40]:
```
def compara(C1,C2):
    k = 0
```

```
            l = len(C1)
            for j in xsrange(l):
                if C2[j]==C1[j]:
                    k += 1
                else:
                    break
            return k
```

```
In [42]: C1 = cifra_log2(1000,2048); C2 = (log(2).n(prec=3048)).str(base=2); compara(C1[2:],C2
```

```
Out[42]: 2035
```

```
In [43]: def primera():
            for k in srange(3,10):
                precis= 2**k
                C1 = cifra_log2(1000,precis)
                C2 = (log(2).n(prec=1000+precis)).str(base=2,no_sci=2)
                print compara(C1[2:],C2[1001:])
```

```
In [44]: primera()
```

```
0
6
23
51
116
243
495
```

```
In [49]: def segunda():
            L = []
            for j in srange(1,30):
                C1 = cifra_log2(1000*j,1024)
                C2 = (log(2).n(prec=1024+1000*j)).str(base=2);
                c = compara(C1[2:],C2[1000*j+1:])
                #print j,c
                L.append(c)
            return L
```

```
In [50]: print segunda()
```

```
[1014, 1007, 1009, 1007, 1001, 1006, 1007, 1004, 1007, 1008, 1006, 1004, 999, 1004, 1004, 1006
```

Estas dos últimas funciones comprueban las dos afirmaciones que se hacen en el enunciado.

```
In [ ]:
```