Computación Cuántica: Parte II. Ejercicio 1

Alberto García Planes
Profesor: Alberto Galindo

2019-2020

1 Ejercicio 1

El código se realizará íntegramente en el lenguaje Python (versión ≥ 3.6). La entrada, proporcionada en el enunciado del problema, será el siguiente diccionario de probabilidades con las probabilidades de ocurrencia que introducimos como sigue.

```
probs={} #We'll store the probabilities in a dictionary with the symbols as keys.
probs["-"]=0.1927;probs["A"]=0.0575;probs["B"]=0.0128;probs["C"]=0.0263;probs["D"]=0.0285;
probs["E"]=0.0912;probs["F"]=0.0173;probs["G"]=0.0133;probs["H"]=0.0313;probs["I"]=0.0599;
probs["J"]=0.0006;probs["K"]=0.0084;probs["L"]=0.0335;probs["M"]=0.0235;probs["N"]=0.0596;
probs["O"]=0.0689;probs["P"]=0.0192;probs["Q"]=0.0008;probs["R"]=0.0508;probs["S"]=0.0567;
probs["T"]=0.0706;probs["U"]=0.0334;probs["V"]=0.0069;probs["W"]=0.0119;probs["X"]=0.0073;
probs["Y"]=0.0164;probs["Z"]=0.0007
```

1.1 Apartado 1

Recordemos que la entropía estaba definida de la siguiente forma: para un alfabeto de n símbolos con probabilidad de ocurrencia p_i con $i = 1, \cdots, n$, esta viene dada por

$$H(X) = -\sum_{i=1}^{n} \log_2(p_i) p_i$$

Calculamos esta entropía con la siguiente λ-función en Python.

```
calculateEntropy = lambda probs: sum([-np.log2(p)*p for _,p in probs.items()])
```

y obtenemos el valor $H(X) = 4.109151806272626$ bits.

Para calcular el codigo Huffman binario seguiremos la siguiente linea. Comenzaremos con un diccionario indexado por los veintiséis símbolos del alfabeto y cuyo contenido son cadenas vacías que irán construyendo las palabras código. Para ello iremos agrupando en cada iteración del algoritmo los dos símbolos (o agrupaciones de símbolos) con menor probabilidad conjunta y los ordenaremos

para que a aquel grupo con el símbolo que antes aparezca en en alfabeto[1] (lo suponemos ordenado según el orden de entrada) se le asigne un caracter 0 en el código y a los del grupo restante un 1. Una vez procesados los dos grupos se eliminan de nuestro espacio de trabajo y se añade un nuevo grupo formado por la concatenación de ambos. La función que hemos implementado para el cálculo es la siguiente.

```
1  def myHuffman2(probs): #This function computes the huffman binary code for a
   ↪  given symbol list alongside their probability, and returns it in a dict
2    # indexed by symbol.
3    current=list(probs.keys()) #We start with a exhaustive list of symbols
4    code={i:"" for i in current} #And the dictionary that will later store the code
     ↪  (now empty).
5
6    while len(current)>=2: #While there are at least two elements left
7      a,b=getMinProbability(probs,current,2) #We get the two elements with minimum
       ↪  probability
8      a,b=swapAccordingTo2(a,b,probs) #And order them to match the convention
       ↪  stated in class and in the provided slides.
9      for char in a: code[char]+="0"  #To the symbols in the first element, append
       ↪  a "0" char to the code of that symbol
10     for char in b: code[char]+="1" #and for the ones in the 2nd, a "1" char
11     current.remove(a); current.remove(b) #Once they're processed, we delete them
       ↪  from the list
12     current.append(a+b) #and append the concatenation of them both to the list
13     #Note: we do not calculate the sum of probabilities for the new element, as
       ↪  we'll recalculate it every time
14     # it's inefficient but we want to keep it simple.
15
16   for k,v in code.items(): code[k]=v[::-1] #Last, for every code, we must reverse
     ↪  it, as we've created it backwards
17   return code #And that's it.
```

Las funciones *getMinProbability* y *swapAccordingTo2* son funciones auxiliares que no hacen más que encontrar los dos grupos del espacio de trabajo con mínima probabilidad y ordenarlos respecto al convenio que hemos mencionado. No las incluimos por economía pero se pueden consultar en el siguiente enlace: https://github.com/albgp/ComputCuanticaUCM.

Esta función, aplicada a la lista de probabilidades dadas nos devuelve el código binario que vemos en la figura 1.

El cálculo de la longitud media (ponderada, tal y como se indica en los apuntes) la calculamos con el siguiente método

[1] Habría multitud de códigos Huffman equivalentes, esto es para adaptarnos a las convenciones de las *slides* de la asignatura.

Símbolo	Probab	HuffmanCod
-	0.1927	00
A	0.0575	0100
B	0.0128	011000
C	0.0263	01101
D	0.0285	10000
E	0.0912	1100
F	0.0173	111000
G	0.0133	011001
H	0.0313	10001
I	0.0599	1001
J	0.0006	1101000000
K	0.0084	1010000
L	0.0335	11101
M	0.0235	110101
N	0.0596	0101
O	0.0689	1011
P	0.0192	111001
Q	0.0008	110100001
R	0.0508	11011
S	0.0567	0111
T	0.0706	1111
U	0.0334	10101
V	0.0069	11010001
W	0.0119	1101001
X	0.0073	1010001
Y	0.0164	101001
Z	0.0007	1101000001

Figure 1: Código Huffman binario calculado.

```
length = lambda probs, codes: sum(len(v)*probs[k] for k,v in codes.items())
  ↪  #Function to compute the "mean length" of a huffman code
```

lo que nos devuelve que la longitud media para el código obtenido es 4.1456, lo que es muy ligeramente superior a la entropía de la distribución de probabilidad. Esto tiene sentido dado que sabemos que no existen códigos sin pérdida con longitud media menor que la entropía ([Goemans, 2013] Teorema 1) pero también sabemos que el código Huffman es óptimo (aunque no necesariamente alcanza el valor de la entropía).

Para comprobar si la desigualdad de Kraft-McMillan se cumple (sabemos de antemano que sí, por el resultado del Lema 1 de [Goemans, 2013] y la construcción arbórea usual del código Huffman, o usando el resultado por el cual la desigualdad de K-McM se cumple si y sólo si el código es de decodificación única [Roweis, 2005]). Usamos el método siguiente

```
tol=1E-12
KMid = lambda base,codes : sum([base**(-len(v)) for _,v in codes.items()])<=1+tol
  ↪  #Returns true if the K-M identity holds for the code "codes" in base "base"
  ↪  (up to floating point precission with tolerance).
```

Esto, aplicado a nuestro código, nos devuelve **True** dado que el lado izquierdo de la desigualdad es 1.0.

1.2 Apartado 2

Veamos ahora cómo calcular el código Huffman ternario. En el caso anterior recordemos que, en cada iteración, obteníamos los dos grupos con menor probabilidad y la asignábamos los caracteres 0 y 1. En esta caso, obtendremos los tres grupos con menor probabilidad (sin más que ordenar la lista de grupos, lo que hasta ahora hemos llamado espacio de trabajo, de acuerdo a la función de cálculo de probabilidad) y les asignaremos, respectivamente y según el convenio que ya hemos mencionado, los caracteres 0, 1 y 2 respectivamente. Una particularidad es que, en la última iteración del método podrán quedar tan solo dos elementos (es decir, puede no haber un grupo al que asignarle el caracter 2) y tendríamos que llevar cuidado para que nuestros métodos auxiliares no fallasen. Por ello, para este caso simplemente usaremos el mismo código que la función para el cómputo del código huffman binario. Queda entonces la siguiente función implementada.

```
def myHuffman3(probs): #Ternary huffman code generator function
  current=list(probs.keys())
  code={i:"" for i in current}


  while len(current)>1:
```

```
    if len(current)==2: #Special case in which there are only two groups of
    ↪  symbols left to process.
      a,b=getMinProbability(probs,current,2) #Which is just the same as the
      ↪  binary treatment of the array
      a,b=swapAccordingTo2(a,b,probs)
      for char in a: code[char]+="0"
      for char in b: code[char]+="1"
      current.remove(a); current.remove(b)
      current.append(a+b)

    else:
      a,b,c=getMinProbability(probs,current,3)
      a,b,c=swapAccordingTo3(a,b,c,probs)
      for char in a: code[char]+="0"
      for char in b: code[char]+="1"
      for char in c: code[char]+="2"
      current.remove(a); current.remove(b); current.remove(c)
      current.append(a+b+c)

  for k,v in code.items():
    code[k]=v[::-1] #And finally, reverse it

  return code
```

Y obtenemos el código Huffman que podemos visualizar en la figura 2.

Con las mismas funciones que ya hemos mostrado (pues las programamos de forma que sean adaptables a diferentes bases) obtenemos que la longitud media es 2.6741000000000006 y el lado izquierdo de la desigualdad de Kraf-McMillan es 0.9999999999999996 (en realidad sabemos que es 1, pero el resultado no es exacto por precisión aritmética flotante), por lo que también se cumple la desigualdad en este caso.

1.3 Apartado 3

La función de codificación es extremadamente simple y la mostramos a continuación

```
encodeMsg=lambda code, msg: "".join([code[char] for char in msg]) #Given a code
↪  and a msg, it returns the encoded msg, C(msg)
```

Con ella codificamos el mensaje dado y obtenemos el mensaje codificado

"0100002201101202001011200101212110010001022002211212201000110220012012211020200112002221000221122110020022112120202001102110210111100210121010202001000120010020210112222120120011121100201200100002020001000201010221121220101022011110011010012002120000221121 2

Símbolo	Probab	HuffmanCod
-	0.1927	00
A	0.0575	010
B	0.0128	1000
C	0.0263	110
D	0.0285	111
E	0.0912	12
F	0.0173	200
G	0.0133	0200
H	0.0313	112
I	0.0599	011
J	0.0006	020100
K	0.0084	1001
L	0.0335	101
M	0.0235	201
N	0.0596	012
O	0.0689	21
P	0.0192	202
Q	0.0008	020101
R	0.0508	021
S	0.0567	022
T	0.0706	22
U	0.0334	102
V	0.0069	02011
W	0.0119	1002
X	0.0073	02012
Y	0.0164	0202
Z	0.0007	020102

Figure 2: Código Huffman ternario calculado.

00112011020011212022220011010101022022”

que, leído en base tres es el siguiente decimal

$$n = 189880498154874410908471072304455370977803974052993577744714297943955583553239505756289235529074177090189772883747055299991539896681082073700.$$

1.4 Apartado 4

La función de decodificación se muestra a continuación. Esta, para cada mensaje, construye iterativamente el mensaje decodificado buscando un *match* al inicio de la secuencia con una palabra código de nuestro código. Recordemos que esto va a funcionar siempre que un mensaje esté bien codificado, dado que el código huffman es instantáneo y biunívoco (de decodificación única).

```
def decodeMsg(code, msg): #Method to decode a msg according to a huffman code for its alphabet
  #Inefficient, but ok, giving up the cool efficient tree structure decomposition.

  #Assuming the input msg corresponds to a code-encoded original, otherwise the algorithm may never end.
  decodedmsg=""
  while msg!="": #While not fully decoded
    for k,v in code.items(): #We look for the matching huffman code at the begining of the remaining sequence
      if msg.startswith(v): #Found it!
        decodedmsg+=k #Append the char to the decoded msg
        msg=msg[len(v):] # and delete the already decoded part from the original seq
        break # Go find the next one
  return decodedmsg
```

Si decodificamos con esta función la cadena dada obtenemos el mensaje siguiente

EL-ENTERO-N-ANTERIOR-ES-DIVISIBLE-AL-MENOS-POR-SIETE-PRIMOS-DISTINTOS

Veamos que esta afirmación es correcta. Para ello primero apliquemos un algoritmo de fuerza bruta para encontrar los divisores primos más pequeños, que se puede visualizar a continuación

```
#Not the coolest algorithm for prime checking, as it works in O(2^(n/2)) in the worst case scenario for a n-bit int, but it's okay for our purposes.
isPrime = lambda n: not any(n%k==0 for k in range(2,int(sqrt(n)-1)+2))
```

```
for i in range(2,10000000):
  if (n%i==0) and isPrime(i):
    print(i)
```

esto comprueba que los divisores primos menores que 10^7 del entero n son

$$\{2, 5, 653, 348527, 551569\}$$

Podemos intentar hallar más de esta forma, pero nuestro esfuerzo ha sido inútil dado que el siguiente divisor primo parece ser extremadamente grande y, ni siquiera los métodos de Brent o el método rho de Pollard implementado en *Sympy* encuentran nada en un tiempo aceptable. Adoptaremos entonces el siguiente enfoque: sea el número m que obtenemos al dividir iterativamente por todos los primos encontrados es compuesto y no es potencia de primo podremos concluir que el número original tenía, al menos, 7 divisores primos distintos. El número m mencionado es

$m = 151262371044142899958200105349142839517999439239432513835740302875266170086611$
$337575158404397890203673622797917709940249650 83$

Afortunadamente, ¡ambas cosas se pueden realizar en tiempo polinomial aceptable! El método para comprobar si es potencia de primo es sencillo. Para comprobarlo notemos que, si $m = p^i$, entonces $i = \log_p m \leq \log_2 m$, por lo que solo tendremos que probar tantos números como número de bits hayan sido necesarios para codificar el número, que en nuestro caso es del orden de 400. Este algoritmo es por tanto, si el bucle interior se comporta en un tiempo polinomial, polinomial en el tamaño de la entrada. Para cada i en el rango mencionado, vemos si existe un p tal que $p^i = m$ (lo comprobamos para los dos valores adyacentes del valor $\lfloor \sqrt[i]{m} \rfloor$ para no pillarnos los dedos con errores de precisión de coma flotante. Si esto se cumple, solo tenemos que ver si este número es primo (el algoritmo AKS era ciertamente ineficiente[2] y usaremos el de Miller-Rabin ([Rabin, 1980]), que explicaremos en breve), que, dado un primo, detecta que lo es con una probabilidad de éxito de $1 - 4^{-k}$, donde hemos elegido en nuestro algoritmo $k = 15$ pues ya era una probabilidad de éxito muy buena.

```
#We'll know, using the MR algorithm, that the number is composite BUT, it may be
↪  a power of a prime, and then we'd have only one prime factor in m.
# To be sure that it has at least two of them, let's check, in low time
↪  complexity if that is the case.
def isPowerOfPrime(n):
  for i in range(2,int(log2(n))+1): #If p^i=n, then i<=log2(n)
    possiblePrime=int(pow(n,1/i)) #Get the integer closest to the power n^(1/i)
    if pow(possiblePrime+1, i)==n: #We assume floating point errors, so we check
    ↪  for x= p-1,p and p+1. If it turns that x^i=n
    #then we've found that n is a power. If x is prime (we know it to incredible
    ↪  precission using MR algorithm), then it's a power of prime
      if MillerRabinTest(possiblePrime+1): return True
```

[2] Aunque este esté en **P**, tiene una complejidad en $\mathcal{O}(n^{21/2})$ ([Agrawal et al., 2004]), lo que hace que fueran necesarias alrededor de 10^{27} operaciones para resolver el caso que nos compete, lo que no es realizable.

```
    elif pow(possiblePrime, i)==n:
      if MillerRabinTest(possiblePrime): return True
    elif pow(possiblePrime-1, i)==n:
      if MillerRabinTest(possiblePrime-1): return True
  return False #Else it's not
```

Hemos visto ya cómo detectar si el número m es potencia de primo, veamos también cómo detectar si es un número compuesto. Pongámonos algo técnicos para comprender cómo funciona el test de primalidad de Miller-Rabin, que usaremos para comprobarlo. Sabemos que, por el pequeño teorema de Fermat para la aritmética modular, para cualquier primo p se cumple que, si $a \in \mathbb{Z}$, $a^p \equiv a \mod p$. Como en este caso $\mathbb{Z}_p$ es un cuerpo, si tenemos que $x^2 = 1 \mod p$, entonces, usando la descomposición $x^2 - 1 = (x-1)(x+1) \equiv 0 \mod p$, como un cuerpo es en particular un dominio de integridad debe cumplirse que $x \equiv \pm 1 \mod p$. De lo que hemos mencionado se sigue que $a^{p-1} \equiv 1 \mod p$, por lo que si $p-1$ es par, $a^{(p-1)/2} \equiv \pm 1 \mod p$. De esto se deduce que si p es primo, para cada $a \in \mathbb{Z}_p^*$ (grupo de los elementos invertibles en $\mathbb{Z}_p$, es decir, coprimos con p) se tiene que si $n - 1 = 2^s r$, entonces o $a^d \equiv 1 \mod p$ o bien, $a^{2^r d} \equiv -1 \mod p$ para algún $r = 0, \cdots, s$. Por contrarecíproco, si un número n no cumple tal razonamiento, debe ser compuesto. Si la cumple nótese que no podemos decir nada. La implementación de esto se muestra a continuación.

```
def MillerRabinTest(n, verbose=False):# Miller-Rabin primality test. To avoid
↪  complications, works well for n>10
  #Returns: False if composite, True if maybe a prime.
    assert n==int(n)

    #We look for the decomposition of n-1=d*2^s with larger s
    s = 0
    d = n-1
    while d%2==0:
        d>>=1 #d//=2 but faster
        s+=1
    assert(2**s * d == n-1)

   #The following conditions hold for primes bc if n is prime, then by Fermat's
    ↪  little theorem, a^(n-1)=1 (mod n). By the definition of prime in a field,
   #and the decomposiion x^2=(x-1)(x+1), if n=2^rd, as calculated, then a^d=1
    ↪  (mod n) or a^(d2^s)=-1. If it is not satisfied by any of the s and d that
    ↪  define the decomposition, then it cannot be a prime.
    def isComposite(a): #Checking if any prime-conditioned constraint holds, then
     ↪  we cannot conclude that a is prime
      # The iterative powering must be done in a modular way (otherwise it'd be
       ↪  exponential in time). Pow allows it.
        if pow(a, d, n) == 1: return False #Equality a^d (mod n) =1 => n may be
         ↪  prime
        for i in range(s): #As s was the larger, we only need to try i-values up
         ↪  to s-1.
```

```
            if pow(a, 2**i * d, n) == n-1: return False #a^(d2^i) mod(n)=n-1=-1 (mod n) => may be prime
        if verbose:
          print("El test de Miller-Rabin ha hallado un compuesto con semilla a={}".format(a))
        return True  #If it does not satisfy any of the above contraints,
        #then, by contrapositive, it can't be prime.

    for i in range(15):# If n is probably prime, number of chances to find the composite answer. Success probability: 1-1/4^(15).
        a = random.randrange(2, n) #Choosing a seed
        if isComposite(a): #If we know for sure that it's not a prime, we return composite
            return False

    return True  #Else, returns probably prime. (Deterministic for negative answers, probabilistic for positive ones)
```

Al ejecutar este método con el número m como entrada obtenemos que para la semilla $a = 374447519246123336641908436307096163554247665016000173875144347198217449826532848580246687509040222472368661457198461071335 3$ no se cumple la condición mencionada, por lo que m no puede ser primo. Como tampoco es potencia de primo, concluimos que existen al menos 7 divisores primos distintos de n (los cinco encontrados directamente y dos más que hemos deducido que deben dividir a m).

1.5 Apartado 5

Partimos de las siguiente matrices, especificadas en el enunciado del problema.

```
H=np.array([
  [0,0,0,0,0,0,0,1,1,1,1,1,1,1,1],
  [0,0,0,1,1,1,1,0,0,0,0,1,1,1,1],
  [0,1,1,0,0,1,1,0,0,1,1,0,0,1,1],
  [1,0,1,0,1,0,1,0,1,0,1,0,1,0,1]])

G=np.array([
  [1,1,0,1,0,0,0,1,0,0,0,0,0,0,1],
  [0,1,0,1,0,0,0,1,0,0,0,0,0,1,0],
  [1,0,0,1,0,0,0,1,0,0,0,0,1,0,0],
  [0,0,0,1,0,0,0,1,0,0,0,1,0,0,0],
  [1,1,0,0,0,0,0,1,0,0,1,0,0,0,0],
  [0,1,0,0,0,0,0,1,0,1,0,0,0,0,0],
  [1,0,0,0,0,0,0,1,1,0,0,0,0,0,0],
  [1,1,0,1,0,0,1,0,0,0,0,0,0,0,0],
  [0,1,0,1,0,1,0,0,0,0,0,0,0,0,0],
```

```
  [1,0,0,1,1,0,0,0,0,0,0,0,0,0,0],
  [1,1,1,0,0,0,0,0,0,0,0,0,0,0,0]])

D="010001010110100010110011111010111100111001111001110011001100\
1111001101111111010001010000000110110110000101101100111 0\
1100111101111000111000010110101011111110011011111101110 0\
0101101111001011010010111010100101101011101011010010101 1\
1101101010101011"
```

En primer lugar deberemos calcular los síndromes de cada sección de 15 caracteres recibida en **D**. Para ello usamos la el código

```
syndromes=[H.dot(v)%2 for v in D] #For each vector v in D, we calculate de
 ↪  syndrome Hv, where the product is performed in F_2~Z_2.
#these will be, in binary, the positions where the transmission errors occurred.
```

Sabemos que las representaciones en binario de dichos síndromes nos indican en qué posición del mensaje se ha producido el error, podemos calcular dichas posiciones y arreglarlas (sumarles un 1 en aritmética modular de módulo 2)

```
binaryArrayToInt = lambda x: int("".join([str(i) for i in x]),2)

Dlimpias=D.copy()

for i, syn in enumerate(syndromes): #For each calculated syndrome
  position=binaryArrayToInt(syn) #Let's take the integer that it represents in
   ↪  Z^4_2
  if position==0:
    continue
  Dlimpias[i][position-1]+=1 #and correct it in Dlimpias. The -1 stands for the
   ↪  possition correction, as python indexes from 0.

Dlimpias%=2 # The operation was in Z_2 so we need to assure that we didn't leave
 ↪  Z_2. This corrects it if necessary
```

El nuevo vector que obtenemos (**Dlimpias**) es

Dlimpias= 010001010110100110110011111010101100111001111000110011001111001001111111
010001000000000110110111000101101100111111001111011110011110000101101011111111100110111
011011100010110101100101101001010101010010110101010101101001010101101101010101010

Podemos ahora eliminar las colimnas de **Dlimpias** que corresponden a bits de corrección de errores, que sabemos que son aquellas cuyo índice es una potencia de dos, por la forma de la matriz

G. También invertimos la secuencia pues observando la forma de G queda claro que esta mapea las posiciones del vector de entrada de forma inversa a las posiciones que no son potencia de dos del mensaje codificado, por ejemplo, la decimoprimera a la tercera, o la décima a la quinta. Una vez hecho esto, tras juntar todos los fragmentos de **Dlimpia** de nuevo en una única secuencia y, tras convertirlo a binario, obtenemos el entero

$k = 16666666666666666666666666666666666666666666666666666661$.

Por último, mostramos el código que se ha utilizado para realizar esta tarea

```
okCols=np.array([2,4,5,6,8,9,10,11,12,13,14]) #Indexed from 0! Slice for slicing, taking out the 2^k-th value for k=0,1,2,3.
#Just bc all power-of-two positions are parity (error correction) bits in Hamming coding.

Dlimpias=np.array([i[okCols][::-1] for i in Dlimpias]) #Take the interesting columns from the vectors in Dlimpias and recover the original msg.


n=binaryArrayToInt(Dlimpias.reshape(nBlocks*11)) #Read it in binary
print("El número codificado en la secuencia original, recuperado con el control de errores basado en códigos Hamming y en base 10 es: {}".format(n))
```

Computación Cuántica: Parte II. Ejercicio 2

Alberto García Planes
Profesor: Alberto Galindo

2019-2020

2 Ejercicio 2

El cálculo lo realizaremos, como en el apartado anterior, mayoritariamente en Python, aunque veremos que necesitaremos otro software más potente para ciertas tareas concretas, que mencionaremos más adelante. En primer lugar tenemos que definir los operadores de espín para sistemas de espin 1/2 y de espín 1.

```
S2x=Rational(1,2)*Matrix([[0,1],[1,0]])
S2y=-Rational(1,2)*I*Matrix([[0,1],[-1,0]])
S2z=Rational(1,2)*Matrix([[1,0],[0,-1]])

S3x=1/sp.sqrt(2)*Matrix([[0,1,0],[1,0,1],[0,1,0]])
S3y=-1/sp.sqrt(2)*I*Matrix([[0,1,0],[-1,0,1],[0,-1,0]])
S3z=Matrix([[1,0,0],[0,0,0],[0,0,-1]])

S2=[S2x,S2y,S2z] #Spin 1/2 operator
S3=[S3x,S3y,S3z] #Spin 1 operator
```

2.1 Apartado 1.1

Vamos a calcular los proyectores P_A, P_B y P_C que se nos piden. El primero de ellos será el proyector sobre el espacio generado (calculado con el producto externo del vector normalizado) por al autovector correspondiente all autovalor positivo (dirección $+Ox$) de la matriz S_x^2, puesto que $(1,0,0)\cdot S^{1/2} = S_x^{1/2}$. De la misma forma tenemos que P_B es el proyector sobre el subespacio generado por el autovector de autovalor positivo de $S_z^{1/2}$. P_C se calcula como el proyector esta vez generado por el autovector con autovalor negativo (puesto que apunta en la dirección contraria) del operador $\frac{1}{3}(\sqrt{8},0,1)\cdot S^1$ donde S^1 es el operador de espín 1. Esto lo podemos calcular con el siguiente código.

```
paBuild=S2x.eigenvects() #Calculate eigenvects for the spin 1/2 matriz in the x
↪  direction
paEigVNorm=paBuild[1][2][0]/paBuild[1][2][0].norm() # take the eigenvector
↪  (normalized) corresponding to the "+" direction (the one with +1/2 eigenval)
```

```
Pa=paEigVNorm*paEigVNorm.T #Calculate the projector

pbBuild=S2z.eigenvects() #Same as before with the z Matrix (points in the +Oz direction)
pbEigVNorm=pbBuild[1][2][0]/pbBuild[1][2][0].norm()
Pb=pbEigVNorm*pbEigVNorm.T

eigvecsnS3=sp.simplify((Rational(1,3)*(sp.sqrt(8)*S3x+S3z)).eigenvects()) #Eigenvectors for S3·n where n=1/3*(sqrt(8),0,1)


eigenvecMinus1=sp.simplify(eigvecsnS3[0][2][0])/eigvecsnS3[0][2][0].norm() #We take the normalized eigenvector corresponding to the eigenvalue -1 (as it points in the opposite direction).
Pc=eigenvecMinus1*eigenvecMinus1.T #and calculate the projector onto the subsepace spanned by this vector. That's Pc.
```

La matriz obtenida de esta operación es la siguiente.

$$\rho(0) = \frac{1}{72}\begin{pmatrix} 1 & -2 & 2 & 0 & 0 & 0 & -1 & 2 & -2 & 0 & 0 & 0 \\ -2 & 4 & -4 & 0 & 0 & 0 & 2 & -4 & 4 & 0 & 0 & 0 \\ 2 & -4 & 4 & 0 & 0 & 0 & -2 & 4 & -4 & 0 & 0 & 0 \\ 0 & 0 & 0 & 3 & -6 & 6 & 0 & 0 & 0 & 3 & -6 & 6 \\ 0 & 0 & 0 & -6 & 12 & -12 & 0 & 0 & 0 & -6 & 12 & -12 \\ 0 & 0 & 0 & 6 & -12 & 12 & 0 & 0 & 0 & 6 & -12 & 12 \\ -1 & 2 & -2 & 0 & 0 & 0 & 1 & -2 & 2 & 0 & 0 & 0 \\ 2 & -4 & 4 & 0 & 0 & 0 & -2 & 4 & -4 & 0 & 0 & 0 \\ -2 & 4 & -4 & 0 & 0 & 0 & 2 & -4 & 4 & 0 & 0 & 0 \\ 0 & 0 & 0 & 3 & -6 & 6 & 0 & 0 & 0 & 3 & -6 & 6 \\ 0 & 0 & 0 & -6 & 12 & -12 & 0 & 0 & 0 & -6 & 12 & -12 \\ 0 & 0 & 0 & 6 & -12 & 12 & 0 & 0 & 0 & 6 & -12 & 12 \end{pmatrix} \tag{1}$$

2.2 Apartado 1.2

Lema 1. *Un estado ρ es puro si y sólo si* $\mathrm{tr}\{\rho^2\} = 1$.

Demostración. Ver [Nielsen and Chuang, 2002]. □

Podemos calcular entonces la traza de $\rho(0)^2$ con el siguiente código

```
rho0Squared = rho0*rho0

print("La traza de rho^2 es: {}={}".format(sp.trace(rho0Squared), sp.trace(rho0Squared).evalf() ))
```

y obtenemos $\mathrm{tr}\{\rho(0)^2\} = \frac{5}{8} < 1$, por lo que ρ no es puro.

2.3 Apartado 1.3

Lema 2. *Sea ρ un estado y $\{\lambda_i\}$ los autovalores de ρ, entonces la entropía de Von-Neumann se puede escribir como*

$$S(\rho) = -\sum_i \lambda_i \log \lambda_i$$

Demostración. Sea el operador unitario U tal que $\rho = U\text{diag}(\lambda_1, \dots, \lambda_n)U^\dagger$, entonces

$$\begin{aligned} S(\rho) &= -\operatorname{tr}\big\{U\text{diag}(\lambda_1, \dots, \lambda_n)U^\dagger \log\big(U\text{diag}(\lambda_1, \dots, \lambda_n)U^\dagger\big)\big\} \\ &= -\operatorname{tr}\left\{U\text{diag}(\lambda_1, \dots, \lambda_n)\underbrace{U^\dagger U}_{1}\log(\text{diag}(\lambda_1, \dots, \lambda_n))U^\dagger\right\} \\ &= -\operatorname{tr}\{\text{diag}(\lambda_1, \dots, \lambda_n)\log(\text{diag}(\lambda_1, \dots, \lambda_n))\} \\ &= -\sum_i \lambda_i \log \lambda_i \end{aligned}$$

□

Con este resultado podemos, simplemente, calcular los autovalores del operador mencionado y computar la entropía clásica (recordemos que redefinimos $0\log 0$ como 0 por ratio de convergencia). El código para realizar esa tarea es el siguiente

```
eigvals=rho0.eigenvals(simplify=True, multiple=True) #The exhaustive (not grouped) list of eigenvalues, to compute the Von Neuman entropy

print("Los autovalores para rho(0) son: {}".format(eigvals))

def vonNeumanEntropy(eigvals): #Compute the entropy
  notNullEigvals=[v for v in eigvals if v!=0] #As we define 0log0 as 0, we can leave out the null values.
  s=0
  for v in notNullEigvals: #sum the entropy for each eigenvalue
    vNum=v.evalf() #We need to turn them into float values (they are symbolic til now)
    s+=-vNum*log2(vNum)
  return s

print("La entropía von-Neuman de rho(0) es: {}".format(vonNeumanEntropy(eigvals)))
```

Este código nos proporciona un valor para la entropía de $S(\rho(0)) = 0.81127812445913$

2.4 Apartado 1.4

Recordemos que $\rho(0)$ estaba definida de la siguiente forma para ciertos estados $\rho^{(i)}$

$$\rho_{ABC}(0) = \frac{3}{4}\rho^{(1)} + \frac{1}{4}\rho^{(2)}$$

Si estos estados $\rho^{(i)}$ son ambos separables en producto de matrices de densidad, entonces habremos puesto $\rho_{ABC}(0)$ como combinación convexa de estados separables, lo que cumple la definición ([Krammer, 2016]) de estado separable.

$$\rho^{(1)} = \frac{1}{18}\begin{pmatrix}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & -2 & 2 & 0 & 0 & 0 & 1 & -2 & 2 \\
0 & 0 & 0 & -2 & 4 & -4 & 0 & 0 & 0 & -2 & 4 & -4 \\
0 & 0 & 0 & 2 & -4 & 4 & 0 & 0 & 0 & 2 & -4 & 4 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & -2 & 2 & 0 & 0 & 0 & 1 & -2 & 2 \\
0 & 0 & 0 & -2 & 4 & -4 & 0 & 0 & 0 & -2 & 4 & -4 \\
0 & 0 & 0 & 2 & -4 & 4 & 0 & 0 & 0 & 2 & -4 & 4
\end{pmatrix}$$

$$\rho^{(2)} = \frac{1}{18}\begin{pmatrix}
1 & -2 & 2 & 0 & 0 & 0 & -1 & 2 & -2 & 0 & 0 & 0 \\
-2 & 4 & -4 & 0 & 0 & 0 & 2 & -4 & 4 & 0 & 0 & 0 \\
2 & -4 & 4 & 0 & 0 & 0 & -2 & 4 & -4 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
-1 & 2 & -2 & 0 & 0 & 0 & 1 & -2 & 2 & 0 & 0 & 0 \\
2 & -4 & 4 & 0 & 0 & 0 & -2 & 4 & -4 & 0 & 0 & 0 \\
-2 & 4 & -4 & 0 & 0 & 0 & 2 & -4 & 4 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{pmatrix}$$

Estos estados son fácilmente separables

$$\rho^{(1)} = \frac{1}{2}\begin{pmatrix}
0 & 0 & 0 & 0 \\
0 & 1 & 0 & 1 \\
0 & 0 & 0 & 0 \\
0 & 1 & 0 & 1
\end{pmatrix} \otimes \frac{1}{9}\begin{pmatrix}
1 & -2 & 2 \\
-2 & 4 & -4 \\
2 & -4 & 4
\end{pmatrix}$$

$$\rho^{(2)} = \frac{1}{2}\begin{pmatrix}
1 & 0 & -1 & 0 \\
0 & 0 & 0 & 0 \\
-1 & 0 & 1 & 0 \\
0 & 0 & 0 & 0
\end{pmatrix} \otimes \frac{1}{9}\begin{pmatrix}
1 & -2 & 2 \\
-2 & 4 & -4 \\
2 & -4 & 4
\end{pmatrix}$$

Lo que nos queda por probar es que podemos descomponer los estados que pertenecen al sistema AB. Veámoslo

$$\rho^{(1)} = \frac{1}{2}\begin{pmatrix}1 & 1\\ 1 & 1\end{pmatrix} \otimes \begin{pmatrix}0 & 0\\ 0 & 1\end{pmatrix} \otimes \frac{1}{9}\begin{pmatrix}1 & -2 & 2\\ -2 & 4 & -4\\ 2 & -4 & 4\end{pmatrix}$$

$$\rho^{(2)} = \frac{1}{2}\begin{pmatrix}1 & -1\\ -1 & 1\end{pmatrix} \otimes \begin{pmatrix}1 & 0\\ 0 & 0\end{pmatrix} \otimes \frac{1}{9}\begin{pmatrix}1 & -2 & 2\\ -2 & 4 & -4\\ 2 & -4 & 4\end{pmatrix}$$

Por lo que el estado es finalmente de la forma

$$\rho_{ABC} = \sum_{i=1,2} c_i \rho_A^{(i)} \otimes \rho_B^{(i)} \otimes \rho_C^{(i)}$$

con $c_1 = \frac{3}{4}$ y $c_2 = \frac{1}{4}$ y las matrices parciales las mostradas en la descomposición, por lo que es una combinación convexa. Además todas las matrices que aparecen son matrices de densidad, lo que comprobamos con el siguiente código.

```
rhoA1=Matrix([[1,1],[1,1]])/2
rhoA2=Matrix([[1,-1],[-1,1]])/2
rhoB2=Matrix([[1,0],[0,0]])
rhoB1=Matrix([[0,0],[0,1]])
rhoC=Matrix([[1,-2,2],[-2,4,-4],[2,-4,4]])/9

for op in [rhoA1,rhoA2,rhoB1,rhoB2,rhoC]:
  assert all(np.array(op.eigenvals(multiple=True))>=0)
  assert (op.trace()==1)
assert rho0==3*tp(rhoA1,rhoB1,rhoC)/4+tp(rhoA2,rhoB2,rhoC)/4
```

2.5 Ejercicio 2

Para este ejercicio debemos empezar definiendo los operadores de espín en el espacio completo de dimensión 12.

```
#Now we compute, from the spin matrices, the spin operators in the whole
↪  12-dimensional hilbert tensor product space.

SA=[tp(S,eye(2),eye(3)) for S in S2] #tp stands for "tensor product".
SB=[tp(eye(2),S,eye(3)) for S in S2]
SC=[tp(eye(2),eye(2),S) for S in S3]
```

2.6 Apartado 2.1

Una vez hemos definido dichos operadores, calcular H es algo directo. Calcularemos los dos sumandos del hamiltoniano por separado mediante el siguiente código.

```
def H(lambdaVal): #Function to compute the symbolic Hamiltonian
  SAplusSB = [m1+m2 for m1,m2 in zip(SA,SB)] #S_A+S_B
  firstSummand = reduce(lambda x,y:x+y, [m1*m3 for m1,m3 in zip(SAplusSB, SC)])
  ↪  #dot prduct with S_C
  #"sum" does not work for whatever reason. Using functional programming with
  ↪  lambda functions

  SAplusSBplusSC = [m1+m2+m3 for m1,m2,m3 in zip(SA,SB,SC)] #S_A+S_B+S_C
  n = [Matrix(12,12,lambda i,j:0), Matrix(12,12,lambda i,j:0), eye(12)] #
  ↪  n=(0,0,1)
  secondSummand = reduce(lambda x,y:x+y ,[m1*m3 for m1,m3 in zip(SAplusSBplusSC,
  ↪  n)]) #dot product with n

  return firstSummand+lambdaVal*secondSummand #Returns the Hamiltonian
```

Esto nos devuelve el hamiltoniano dependiente del parámetro λ que podemos ver a continuación.

$$H(\lambda) = \begin{pmatrix} 2\lambda+1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & \lambda & 0 & \frac{\sqrt{2}}{2} & 0 & 0 & \frac{\sqrt{2}}{2} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & \frac{\sqrt{2}}{2} & 0 & 0 & \frac{\sqrt{2}}{2} & 0 & 0 & 0 & 0 \\ 0 & \frac{\sqrt{2}}{2} & 0 & \lambda & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{\sqrt{2}}{2} & 0 & 0 & 0 & 0 & 0 & 0 & \frac{\sqrt{2}}{2} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -\lambda & 0 & 0 & 0 & 0 & \frac{\sqrt{2}}{2} & 0 \\ 0 & \frac{\sqrt{2}}{2} & 0 & 0 & 0 & 0 & \lambda & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{\sqrt{2}}{2} & 0 & 0 & 0 & 0 & 0 & 0 & \frac{\sqrt{2}}{2} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -\lambda & 0 & \frac{\sqrt{2}}{2} & 0 \\ 0 & 0 & 0 & 0 & \frac{\sqrt{2}}{2} & 0 & 0 & \frac{\sqrt{2}}{2} & 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & \frac{\sqrt{2}}{2} & 0 & 0 & \frac{\sqrt{2}}{2} & 0 & -\lambda & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1-2\lambda \end{pmatrix}$$

2.7 Apartado 2.2

El espectro vuelve a ser algo trivial sin más que usar la función de la librería Sympy que calcula los autovalores de dicho hamiltoniano. Lo calculamos de la forma que sigue.

```
H=H(l) #Compute and store the Hamiltonian just built.
print("Los autovalores del hamiltoniano son:
↪  {}".format(sp.simplify(H).eigenvals()))
```

Tras esto, obtenemos el espectro

$$\sigma(H(\lambda)) = \{2\lambda+1, 1, \lambda+1, \lambda, \lambda-1, -1, 1-\lambda, -\lambda, -\lambda-1, -2, 1-2\lambda, 0\}$$

2.8 Apartado 2.3

Definimos una función U, que podemos visualizar en el siguiente fragmento de código, que calcula la exponencial de la matriz $cH(\lambda)$ donde c es una constante simbólica que más adelante sustituiremos por $-it$ para obtener el operador de evolución.

```
def U(c,l, H):
  return sp.simplify(sp.exp(c*sp.simplify(H)))

U=U(t,l, H) #When we introduce the complex constant in the function above, it
 ↪  crashes
#so we do it for an arbitrary constant t. will replace t by -t*i later.
```

Esto nos da un operador de evolución cuyas primeras cinco columnas (por espacio) se muestran en la siguiente matriz

$$U(t,\lambda) = \begin{bmatrix}
e^{-it(2\lambda+1)} & 0 & 0 & 0 & 0 \\
0 & e^{-i\lambda t}\cos(t) & 0 & \frac{\sqrt{2}\left(1-e^{2it}\right)e^{-it(\lambda+1)}}{4} & 0 \\
0 & 0 & \frac{e^{2it}}{3}+\frac{e^{it}}{2}+\frac{e^{-it}}{6} & 0 & \frac{\sqrt{2}\left(1-e^{3it}\right)e^{-it}}{6} \\
0 & \frac{\sqrt{2}\left(1-e^{2it}\right)e^{-it(\lambda+1)}}{4} & 0 & \frac{(2\cos(t)+2)e^{-i\lambda t}}{4} & 0 \\
0 & 0 & \frac{\sqrt{2}\left(1-e^{3it}\right)e^{-it}}{6} & 0 & \frac{e^{2it}}{6}+\frac{1}{2}+\frac{e^{-it}}{3} \\
0 & 0 & 0 & 0 & 0 \\
0 & \frac{\sqrt{2}\left(1-e^{2it}\right)e^{-it(\lambda+1)}}{4} & 0 & \frac{(2\cos(t)-2)e^{-i\lambda t}}{4} & 0 \\
0 & 0 & \frac{\sqrt{2}\left(1-e^{3it}\right)e^{-it}}{6} & 0 & \frac{e^{2it}}{6}-\frac{1}{2}+\frac{e^{-it}}{3} \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & \frac{e^{2it}}{3}-\frac{e^{it}}{2}+\frac{e^{-it}}{6} & 0 & \frac{\sqrt{2}\left(1-e^{3it}\right)e^{-it}}{6} \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0
\end{bmatrix}$$

Para visualizar lo que ocurre, dibujemos en la figura 3 la norma de $U_1(0) - U_1(t)$ para valores de t en el intervalo $[0, 8\pi]$, lo que nos dará cierta intuición sobre el comportamiento de $U_1(t)$.

Vemos que la norma se anula para valores de t cercanos a múltiplos de 2π y que las curvas de evolución son exactamente iguales en cada uno de los intervalos $[2k\pi, 2(k+1)\pi]$, lo que nos hace sospechar con cierta seguridad que el operador es cíclico de periodo 2π. Esto se puede comprobar analíticamente viendo que cada término es cíclico con periodo 2π, puesto que, por ejemplo, $e^{(2\pi+t)i} = e^{it}$ y $\cos(2\pi + t) = \cos t$, el cálculo sistemático para cada término se ha asegurando que $U(t,1) - U(t+2\pi, 1) = 0\ \forall t$, con el siguiente código

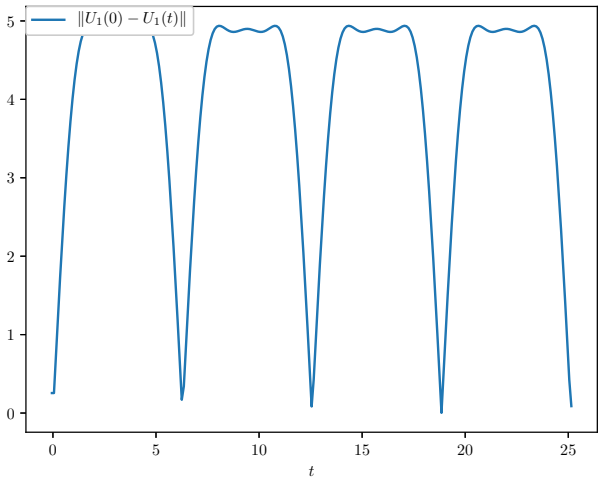

Figure 3: Evolución de la magnitud $\|U_1(t) - U_1(0)\|$

```
assert sp.simplify(sp.simplify(Ut1.subs(t,t+2*sp.pi)-Ut1))==Matrix(12,12,lambda
↪  i,j:0)
```

Sabemos entonces que el operador es periódico de periodo 2π pero, ¿podría ser el operador periódico de orden más pequeño (que divide exactamente el intervalo 2π en partes iguales)? La respuesta es no y la razón es la gráfica 3 en la que hemos visto que $\|U(k,1) - U(0,1)\| > 0 \ \forall k \in (0, 2\pi)$. El periodo es, por tanto, 2π.

2.9 Apartado 2.4

En este ejercicio debemos calcular la distancia en traza y la fidelidad entre los estados $\rho(0)$ y $\rho(t)$, para ello obtendremos de forma directa ambos estados para cada instante t entre 0 y 2π de forma numérica y dibujaremos las gráficas. Recordemos que ambas magnitudes vienen definidas de la siguiente forma.

- **Distancia en traza**: $d(\rho, \sigma) = \frac{1}{2} \operatorname{tr}\left\{\sqrt{(\rho - \sigma)^2}\right\}$
- **Fidelidad**[3]: $F(\rho, \sigma) = \operatorname{tr}\left\{\sqrt{\sqrt{\rho}\sigma\sqrt{\rho}}\right\}$

El código para dicho cálculo se puede encontrar a continuación y la gráfica pedida se encuentra en la figura 4.

[3]Elegimos esta definición de fidelidad, que aparece en el [Nielsen and Chuang, 2002], frente a la misma elevada al cuadrado, que se puede encontrar también en multitud de textos.

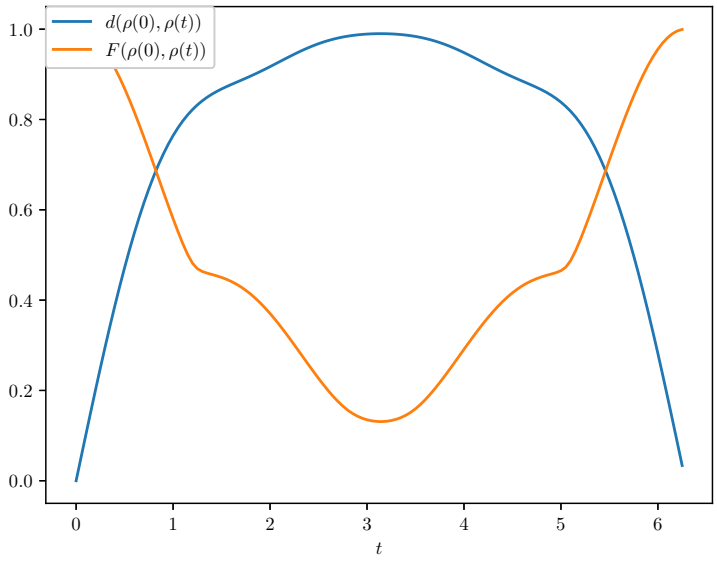

Figure 4: Gráfica para la evolución de la distancia en traza y la fidelidad de $\rho(0)$ y $\rho(t)$ entre 0 y 2π.

```
evOp= lambda t0: np.array(Ut1.subs(t,t0)).astype(np.complex64)

def traceDistance(rho1, rho2):
  x=rho1-rho2
  xConjugate=x.conj().T
  return 0.5*np.trace(sqrtm(xConjugate.dot(x)))

def fidelity(rho1,rho2):
  rho1Sqrt= sqrtm(rho1)
  return np.trace(sqrtm(rho1Sqrt.dot(rho2).dot(rho1Sqrt)))

def plotEvolution(evolutionOperator, T, stepsize, initialState):
  t = np.arange(0.0, T, stepsize)
  initState=np.array(initialState).astype(np.complex128)
  traceDistances=[]
  fidelities=[]
  print(t)
  for instant in t:
    Ut=evolutionOperator(instant)
    currentState=Ut.dot(initialState).dot(Ut.conj().T)
```

```
    currentState=np.array(currentState).astype(np.complex128)
    traceDistances.append(traceDistance(initState,currentState))
    fidelities.append(fidelity(initState,currentState))

  plt.clf()
  plt.plot(t,traceDistances, label='$d(\\rho(0),\\rho(t)$')
  plt.plot(t,fidelities, label='$F(\\rho(0),\\rho(t)$')
  plt.legend(loc='upper left', borderaxespad=0.)
  plt.xlabel("$t$")
  plt.savefig('DistFid.eps', format='eps')
  plt.show()

plotEvolution(evOp, 2*np.pi, 0.1, rho0)
```

Si calculamos el máximo de la distancia en traza y el mínimo de la fidelidad en el intervalo $(0, 2\pi)$ obtenemos que ambos valores se encuentran cuando $t = \pi$, siendo el mínimo del valor de fidelidad 0.130975 y el máximo de la distancia en traza 0.990066, al menos hasta la precisión que *scipy* ofrece por defecto para los métodos de optimización. El código utilizado es el siguiente.

```
initialState=rho0
initState=np.array(rho0).astype(np.complex128)
evolutionOperator=evOp

def fidRho(t):
  Ut=evolutionOperator(t)
  currentState=Ut.dot(initialState).dot(Ut.conj().T)
  currentState=np.array(currentState).astype(np.complex128)
  return fidelity(initState,currentState)

def minusTraceDrho(t):
  Ut=evolutionOperator(t)
  currentState=Ut.dot(initialState).dot(Ut.conj().T)
  currentState=np.array(currentState).astype(np.complex128)
  return -traceDistance(initState,currentState)

res1=fmin(fidRho, 3)
res2=fmin(minusTraceDrho, 3)
```

3 Ejercicio 3

El mecanismo teórico para calcular los operadores de Kraus para la evolución del subsistema es sencillo. En primer lugar vemos que, como la evolución nosotros la hemos definido para el sistema ABC en completo, para hallar el estado $\rho_C(t)$ deberemos hallar la evolución del estado inicial $\rho_{ABC}(0)$ y calcular la traza para el sistema AB, es decir

$$\rho_C(t) = \mathrm{tr}_{AB}(U(t)\rho_{ABC}(0)U^\dagger(t)) = \mathrm{tr}_A\,\mathrm{tr}_B(U(t)\rho_{ABC}(0)U^\dagger(t))$$

Como nuestro estado inicial se puede poner, como ya hemos visto, como $\rho_{AB}(0)\otimes\rho_C(0)$, escribimos $\rho_C(t)$ calculando las trazas contrayendo con una base de A y B como

$$\rho_C(t) = \sum_{m,n=0}^{1} \langle m|_A \langle n|_B U(t)\rho_{AB}(0)\otimes\rho_C(0)U^\dagger(t)\,|m\rangle_A\,|n\rangle_B$$

Descompongamos ahora por simplicidad el estado $\rho_{AB}(0)$ en su forma diagonal

$$\rho_{AB}(0) = \sum_k p_k\,|v_k\rangle\langle v_k|$$

Llegando a la expresión de evolución para $\rho_C(t)$

$$\rho_C(t) = \sum_{m,n,k=0}^{1} p_k \langle m|_A \langle n|_B U(t)\,|v_k\rangle\,\rho_C(0)\,\langle v_k|\,U^\dagger(t)\,|m\rangle_A\,|n\rangle_B$$

Esto nos motiva entonces a escribir los operadores de Kraus, que denotaremos $K_{mnk}(t)$ como sigue

$$K_{mnk}(t) = \sqrt{p_k}\,\langle m|\,\langle n|\,U(t)\,|v_k\rangle$$

por lo que

$$\rho_C(t) = \sum_{m,n,k} K_{mnk}(t)\rho_C(0)K^\dagger_{mnk}(t)$$

Calculemos entonces estos operadores. Para ello deberemos encontrar primero la forma de descomponer el estado $\rho_{AB}(0)$ de la forma mencionada, lo que conseguiremos sin más que calcular los autovectores normalizados y ver cuáles contribuyen al estado, que tan solo serán dos, por lo que el índice k recorrerá los valores 0 y 1. Una vez que descomponemos $U(t)$ como una matriz de bloques 4×4 basta calcular, para cada una, su contracción con $\langle m,n| = \langle m|\otimes\langle n|$ y $|v_k\rangle$, obteniendo un valor para cada matriz, que forma la matriz $3\otimes3$ que buscamos. El código con el que se ha realizado esta tarea es el siguiente

```
rho_AB=Rational(3,4)*tp(Pa, eye(2)-Pb)+Rational(1,4)*tp(eye(2)-Pa,Pb)
rho_C=Pb

rho_AB_eigvecs=[eiv[2][0]/eiv[2][0].norm() for eiv in rho_AB.eigenvects()]
rho_AB_eigvals=[i for i in rho_AB.eigenvals(multiple=True) if i!=0 ]
print(rho_AB_eigvals)


def createKrausOperator(m,n,v, eigvecs, eigvals, evOp):
  ket_n= sp.Matrix([[1],[0]]) if n==0 else sp.Matrix([[0],[1]])
  ket_m= sp.Matrix([[1],[0]]) if m==0 else sp.Matrix([[0],[1]])
```

```
  ket_mn = tp(ket_m,ket_n)
  Kop=sp.Matrix(3,3,lambda i,j:0)


  for i in range(3):
    for j in range(3):
      subMatrix_ij=evOp[4*i:4*i+4, 4*j:4*j+4]
      Kop[i,j]=sp.simplify(sp.simplify(ket_mn.T*subMatrix_ij*eigvecs[v]))
  Kop=sp.sqrt(eigvals[v])*Kop


  def K(tval):
    return Kop.subs({t0:tval})

  return Kop, K

cumsum=sp.Matrix(3,3,lambda i,j:0)
for m in range(2):
  for n in range(2):
    for v in range(2):
      Kop, K = createKrausOperator(m,n,v,rho_AB_eigvecs, rho_AB_eigvals, Ut1)
      cumsum+=Dagger(Kop)*Kop
      print("K_{},{},{}={}".format(m,n,v,latex(Kop)))
```

y obtenemos los siguientes operadores de Kraus, que mostramos a continuación.

$$K_{000}(t) = \begin{pmatrix} \frac{\sqrt{2}e^{-3it}}{4} & 0 & 0 \\ \frac{\left(1-e^{3it}\right)e^{-it}}{12} & \frac{\sqrt{2}\left(\left(e^{2it}+3\right)e^{it}+2\right)e^{-it}}{24} & 0 \\ 0 & 0 & \frac{\sqrt{2}\left(e^{2it}+2e^{it}+1\right)}{16} - \frac{e^{2it}}{8} + \frac{1}{8} \end{pmatrix}$$

$$K_{001}(t) = \begin{pmatrix} -\frac{\sqrt{6}e^{-3it}}{4} & 0 & 0 \\ \frac{\sqrt{3}\left(1-e^{3it}\right)e^{-it}}{12} & -\frac{\sqrt{6}\left(\left(e^{2it}+3\right)e^{it}+2\right)e^{-it}}{24} & 0 \\ 0 & 0 & \frac{\sqrt{3}\left(-\frac{\sqrt{2}\left(e^{2it}+2e^{it}+1\right)}{8} - \frac{e^{2it}}{4} + \frac{1}{4}\right)}{2} \end{pmatrix}$$

$$K_{010}(t) = \begin{pmatrix} 0 & -\frac{1}{8} + \frac{e^{-2it}}{8} & 0 \\ 0 & 0 & \frac{\sqrt{2}\left(e^{2it}-2e^{it}+1\right)}{16} - \frac{e^{2it}}{8} + \frac{1}{8} \\ \frac{\sqrt{2}\left(\left(2e^{it}-3\right)e^{2it}+1\right)e^{-it}}{24} & \frac{\left(1-e^{3it}\right)e^{-it}}{12} & 0 \end{pmatrix}$$

$$K_{011}(t) = \begin{pmatrix} 0 & \frac{\sqrt{3}\left(-\frac{1}{4}+\frac{e^{-2it}}{4}\right)}{2} & 0 \\ 0 & 0 & \frac{\sqrt{3}\left(\frac{\sqrt{2}\left(-e^{2it}+2e^{it}-1\right)}{8}-\frac{e^{2it}}{4}+\frac{1}{4}\right)}{2} \\ \frac{\sqrt{6}\left(\left(2e^{it}-3\right)e^{2it}+1\right)e^{-it}}{24} & \frac{\sqrt{3}\left(e^{3it}-1\right)e^{-it}}{12} & 0 \end{pmatrix}$$

$$K_{100}(t) = \begin{pmatrix} \frac{\sqrt{2}\left(\left(2e^{it}+3\right)e^{2it}+1\right)e^{-it}}{24} & \frac{\left(1-e^{3it}\right)e^{-it}}{12} & 0 \\ 0 & \frac{\sqrt{2}(\cos(t)+1)e^{-it}}{8} & 0 \\ 0 & 0 & \frac{\sqrt{2}\left(e^{2it}+1\right)}{8}-\frac{e^{2it}}{8}+\frac{1}{8} \end{pmatrix}$$

$$K_{101}(t) = \begin{pmatrix} \frac{\sqrt{6}\left(\left(2e^{it}+3\right)e^{2it}+1\right)e^{-it}}{24} & \frac{\sqrt{3}\left(e^{3it}-1\right)e^{-it}}{12} & 0 \\ 0 & \frac{\sqrt{6}(\cos(t)+1)e^{-it}}{8} & 0 \\ 0 & 0 & \frac{\sqrt{3}\left(\frac{\sqrt{2}\left(e^{2it}+1\right)}{4}+\frac{e^{2it}}{4}-\frac{1}{4}\right)}{2} \end{pmatrix}$$

$$K_{110}(t) = \begin{pmatrix} 0 & \frac{\sqrt{2}(\cos(t)-1)e^{-it}}{8} & 0 \\ \frac{\left(1-e^{3it}\right)e^{-it}}{12} & \frac{\sqrt{2}\left(\left(e^{2it}-3\right)e^{it}+2\right)e^{-it}}{24} & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

$$K_{111}(t) = \begin{pmatrix} 0 & \frac{\sqrt{6}(\cos(t)-1)e^{-it}}{8} & 0 \\ \frac{\sqrt{3}\left(1-e^{3it}\right)e^{-it}}{12} & -\frac{\sqrt{6}\left(\left(e^{2it}-3\right)e^{it}+2\right)e^{-it}}{24} & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

Por último comprobamos que se cumple

$$\sum_{m,n,k=0}^{1} K_{mnk}^{\dagger} K_{mnk} = \mathbb{1}$$

```
assert sp.simplify(sp.simplify(cumsum))==eye(3)
```

4 Ejercicio 4

Contruyamos en primer lugar el operador X a partir de las matrices de espín.

```
def buildX(): #Construimos la forma matricial del operador X
  #X=Y·Y-Z·n
  Y=[S_a+S_b-S_c for S_a,S_b,S_c in zip(SA,SB,SC)]
  Z=[S_a+S_b+S_c for S_a,S_b,S_c in zip(SA,SB,SC)]
  return reduce(lambda x,y:x+y, [y*y for y in Y])+Z[2]
```

```
6
7 X=buildX()
```

$$X = \begin{bmatrix} 4 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 5 & 0 & -\sqrt{2} & 0 & 0 & -\sqrt{2} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 6 & 0 & -\sqrt{2} & 0 & 0 & -\sqrt{2} & 0 & 0 & 0 & 0 \\ 0 & -\sqrt{2} & 0 & 4 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -\sqrt{2} & 0 & 3 & 0 & 0 & 1 & 0 & -\sqrt{2} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 2 & 0 & 0 & 1 & 0 & -\sqrt{2} & 0 \\ 0 & -\sqrt{2} & 0 & 1 & 0 & 0 & 4 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -\sqrt{2} & 0 & 1 & 0 & 0 & 3 & 0 & -\sqrt{2} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 2 & 0 & -\sqrt{2} & 0 \\ 0 & 0 & 0 & 0 & -\sqrt{2} & 0 & 0 & -\sqrt{2} & 0 & 6 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -\sqrt{2} & 0 & 0 & -\sqrt{2} & 0 & 3 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

4.1 Apartado 4.1

Sabemos que la probabilidad de medida sobre un estado ρ de un valor x del conjunto de resultados posibles viene dada por la expresión $p_x = \text{tr}\{\rho P_x\}$, donde P_x es el proyector sobre el subespacio generado por los autovectores unitarios de autovalor x. Para resolver el ejercicio calcularemos los autovalores del operador X, junto a sus autovectores que nos darán los proyectores necesarios para calcular las probabilidades. Cabe mencionar que cuando un autovalor tiene multiplicidad mayor que uno, los autovectores asociados a tal valor que nos devuelve el programa no son necesariamente ortogonales, por lo que no podríamos descomponer el proyector en suma de proyectores asociados a cada autovector. Por esta razón normalizamos con Gram-Schmidt tras obtener los autovectores. Si de esta forma obtenemos un conjunto ortonormal $\{P_1^x, \cdots P_n^x\}$ entonces sabemos que $P_x = P_1^x + \cdots + P_n^x$.

```
1 class SpectralProjector:
2   def __init__(self, val, vecs):
3     self.vecs=GramSchmidt(vecs)
4     self.val=val
5     self.projs=[]
6     for i, vec in enumerate(self.vecs):
7       normalizedProj=vec/vec.norm()
8       self.projs.append(normalizedProj*normalizedProj.T)
9
10  def rhoP(self, rho):
11    return reduce(lambda x,y:x+y, [rho*vec for vec in self.projs])
12
13  def probability(self, rho):
14    return np.trace(self.rhoP(rho))
```

```
15
16  def PrhoP(self,rho):
17    return sp.simplify(reduce(lambda x,y:x+y, [vec*rho*vec for vec in
      ↪  self.projs]))
18
19  def getVal(self):
20    return self.val
21
22 eigvals=X.eigenvals(simplify=True, multiple=True)
23
24 print("Los posibles resultados de la medida en X son: {}".format(eigvals))
25
26 t0 = sp.symbols("t_0", real=True)
27 #global_assumptions.add(Q.real(t0))
28
29 Ut01=Ut1.subs(t,t0)
30
31 eigvects = X.eigenvects()
32 spectralProjectors=[]
33
34 for eigval, multiplicity, vecs in eigvects:
35   spectralProjectors.append(SpectralProjector(eigval, vecs))
36
37 rhot0=Ut01*rho0*Dagger(Ut01)
38
39 t0Vals=np.arange(0,2*np.pi, 0.01)
40 s=0
41 pl=[[] for i in spectralProjectors]
42
43 for i, spr in enumerate(spectralProjectors):
44   #print(i)
45   c=0
46   prob=sp.simplify(sp.simplify(spr.probability(rhot0)))
47   #print("Probabilidad del autovalor {}: {}".format(spr.getVal(),
     ↪  latex(sp.simplify(sp.simplify(prob))) ))
48   print("p_{}=&{}\\\\".format(spr.getVal(), latex(prob) ))
49
50   for t0Val in t0Vals:
51     c+=1
52     if c%100==0:
53       pass
54       #print(c)
55     px=sp.re(prob.subs(t0,t0Val).evalf())
56     #print("Probabilidad para el autovalor {}: {}".format(spr.getVal(), px))
57     pl[i].append(px)
```

Obtenemos los autovalores de X, es decir, los valores que puede tomar la medida, que expresamos en la forma "autovalor : multiplicidad" en la siguiente ecuación

$$\{8:1, 7:1, 6:1, 5:1, 4:1, 3:2, 2:2, 1:2, 0:1\}$$

Una vez obtenidos estos autovalores y ayudándonos de la clase SpectralProjector que hemos definido calculamos las probabilidades de obtención de cada uno de los autovalores como ya hemos mencionado (denotamos la probabilidad de obtener el autovalor x como p_x), que se pueden visualizar a continuación. Cabe remarcar que aunque no pongamos dependencia explícita de t en estas funciones, en realidad, a pesar de ser constantes, estas probabilidades son funciones de t que, en otro caso, podrían no ser constantes.

$$\begin{array}{lllll} p_0 = \frac{1}{72} & p_1 = \frac{\sqrt{2}}{72} + \frac{5}{72} & p_2 = \frac{29}{144} - \frac{\sqrt{2}}{108} & p_3 = \frac{1}{4} - \frac{\sqrt{2}}{12} & \\ p_4 = \frac{1}{6} & p_5 = \frac{1}{24} - \frac{\sqrt{2}}{72} & p_6 = \frac{7}{144} & p_7 = \frac{\sqrt{2}}{12} + \frac{5}{36} & p_8 = \frac{\sqrt{2}}{108} + \frac{5}{72} \end{array}$$

Además, es elemental ver que

$$\sum_{i \in \text{eigenvalues}} p_i = 1$$

lo que nos hace tener confianza en las probabilidades obtenidas.

4.2 Apartado 4.2

El estado $\hat{\rho}(t_0)$ se calcula fácilmente con el siguiente código

```
rho_hat = reduce(lambda x,y:x+y, [spec.PrhoP(rhot0) for spec in
  ↪  spectralProjectors ] )
rho_hat=sp.simplify(sp.simplify(rho_hat))
print(latex(rho_hat))
```

Podemos mostrar solo una pequeña porción del estado (por espacio), por lo que elegimos mostrar las primeras cinco columnas:

$$\hat{\rho}(t_0) = \begin{pmatrix} \frac{1}{72} & 0 & 0 & 0 & 0 \\ 0 & -\frac{\sqrt{2}\cos(t_0)}{648} + \frac{\cos(t_0)}{648} - \frac{\sqrt{2}}{324} + \frac{25}{648} & 0 & -\frac{5\cos(t_0)}{648} + \frac{5\sqrt{2}\cos(t_0)}{1296} - \frac{\sqrt{2}}{1296} + \frac{1}{81} & 0 \\ 0 & 0 & \frac{\sqrt{2}}{648} + \frac{1}{16} & 0 & -\frac{1}{162} + \frac{\sqrt{2}}{288} \\ 0 & -\frac{5\cos(t_0)}{648} + \frac{5\sqrt{2}\cos(t_0)}{1296} - \frac{\sqrt{2}}{1296} + \frac{1}{81} & 0 & -\frac{7\sqrt{2}\cos(t_0)}{1296} + \frac{7\cos(t_0)}{1296} + \frac{\sqrt{2}}{324} + \frac{49}{1296} & 0 \\ 0 & 0 & -\frac{1}{162} + \frac{\sqrt{2}}{288} & 0 & \frac{7}{72} - \frac{\sqrt{2}}{648} \\ 0 & 0 & 0 & 0 & 0 \\ 0 & -\frac{\sqrt{2}\cos(t_0)}{432} + \frac{\cos(t_0)}{216} - \frac{\sqrt{2}}{432} + \frac{1}{108} & 0 & -\frac{\sqrt{2}\cos(t_0)}{432} + \frac{\cos(t_0)}{432} - \frac{\sqrt{2}}{216} + \frac{1}{432} & 0 \\ 0 & 0 & -\frac{1}{162} + \frac{\sqrt{2}}{288} & 0 & -\frac{1}{72} - \frac{\sqrt{2}}{648} \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{\sqrt{2}}{648} + \frac{1}{72} & 0 & -\frac{1}{162} + \frac{\sqrt{2}}{288} \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

4.3 Apartado 4.3

Con la misma fórmula que ya hemos utilizado en el apartado 1.3 podemos hallar la entropía del estado $\rho_{ABC}(1)$:

$$S(\rho_{ABC}(1)) = 0.81127812445 \text{ bits}$$

Podemos ver que esta entropía es la misma que encontramos para $\rho_{ABC}(0)$ en el apartado 1.3. Esto tiene sentido ya que $\rho_{ABC}(1)$ se ha obtenido de $\rho_{ABC}(0)$ mediante una transformación unitaria y, usando la propiedad cíclica de la traza es sencillo ver que la expresión $-\operatorname{tr}(\rho \log \rho)$ es invariante a transformaciones unitarias.

La entropía Von-Neumann para el estado $\hat{\rho}(1)$ es, calculada numéricamente, la siguiente

$$S(\hat{\rho}) = 3.1568354875353077 \text{ bits}$$

Claramente la entropía de $\hat{\rho}(1)$ es mayor que la entropía de $\rho_{ABC}(1)$. Esto es teóricamente consistente pues sabemos que (teorema 11.9 del [Nielsen and Chuang, 2002]) que la entropía de un estado es no decreciente bajo medidas proyectivas como las realizadas al calcular el estado del sistema tras medidas no selectivas. Además, este resultado nos dice que $\rho_{ABC}(1) \neq \hat{\rho}(1)$, puesto que en este caso sería necesario que ambos valores para las entropías coincidiesen. El código para este ejercicio es el siguiente.

```
def vonNeumanEntropy(eigvals): #Compute the entropy
  notNullEigvals=[v.real for v in eigvals if v>1e-7] #As we define 0log0 as 0, we
  ↪  can leave out the null values.
  s=0
  for v in notNullEigvals: #sum the entropy for each eigenvalue
    s+=-v*log2(v)
  return s

w,v = np.linalg.eig(np.array(rho_hat.subs(t0,1)).astype(np.complex128))
print("La entropía Von-Neumann es:{} ".format(vonNeumanEntropy(w)))

rho1=rhot0.subs(t0,1)
rho1=np.array(rho1).astype(np.complex128)
w, v = np.linalg.eig(rho1)
print("La entropía Von-Neumann es:{} ".format(vonNeumanEntropy(w)))
```

5 Ejercicio 5

Para este ejercicio, lo primero que necesitamos en un mecanismo para calcular σ_{AB}, es decir, un mecanismo para calcular la traza parcial de ρ_{ABC} sobre el subsistema C. En este caso, dado que no nos proporciona una ventaja significativa trabajar con valores simbólicos y queremos eficiencia en los cálculos usaremos matrices numéricas. El cálculo de la traza parcial sobre el sistema de

dimensión 3 pasará entonces por descomponer estas matrices numéricas en 4×4 bloques de 3×3 y realizar la traza sobre cada uno de ellos. Esto se puede realizar de la siguiente forma

```
def decomposeProductTensor(mat, dims): #Decompose a nmxnm matrix into a nxnxmxm
↪  matrix. Matrix, n and m as the function input.
  a,b=dims
  return mat.reshape([a, b, a, b]) #Reshapes it an returns it

def myPartialTrace4x3in2ndSpace(mat): #Calculate the partial trace over the 3rd
↪  3x3 system.
  reshaped_matrix = decomposeProductTensor(mat, [4,3]) #Decompose it
  tracedMatrix= np.einsum('jiki->jk', reshaped_matrix) #Sums over repeated
  ↪  indices, si calculate the trace on the 2nd subsystem. (Returns a 4by4
  ↪  density matrix of the AB system)
  return tracedMatrix
```

Obtenemos hora el estado $\rho_{ABC} = \rho(\pi)$ y, con las funciones ya presentadas, el estado σ_{AB} calculando la traza parcial respecto de C. Para ver si los estados están entrelazados aplicaremos el criterio de Peres-Horodecki ([Peres, 1996], [Horodecki et al., 1996]), en el que si las traspuestas parciales respecto de uno de sus subsistemas tiene autovalores negativos, entonces estos estados no pueden ser estados separables. Esto se deriva del hecho de que, si lo fuesen, podríamos expresarlos como combinación convexa de productos tensoriales de matrices densidad y, por tanto, al aplicar la transpuesta parcial a cada uno de estos subsistemas obtendríamos una combinación lineal de productos tensoriales donde tan solo cambia el elemento del producto que corresponde al sistema respecto al cual estamos calculando la traspuesta parcial. Como esta operación da, en cada término, una matriz definida positiva (puesto que las matrices de densidad lo son y la traspuesta lo preserva de forma directa), los autovalores deberán ser, a la fuerza, no negativos. Si este no es el caso, por contrarecíproco, el estado no podrá ser separable. Veamos cómo podemos calcular estas traspuestas parciales.

La función que genera traspuestas parciales es muy similar a la ya presentada para la traza parcial. Primero descompondremos la matriz de entrada en bloques y aplicaremos la traspuesta a cada una de las submatrices bloque. Esto, en notación de matriz tetradimensional es mapear el elemento a_{ijkl} al elemento a_{ijlk}, lo que será ligeramente diferente en nuestro código por la forma en la que numpy indexa matrices bloque (deberemos hacer $a_{ijkl} \to a_{ilkj}$). El código es el siguiente.

```
def myPartialTraspose4x3in2ndSpace(mat): #Calculate the partial traspose over the
↪  3rd (C) 3x3 system.
  reshaped_matrix = decomposeProductTensor(mat, [4,3]) #Decompose it
  trMatrix= np.einsum('ijkl->ilkj', reshaped_matrix)
  return trMatrix.reshape([12,12])

def myPartialTraspose2x2in2ndSpace(mat): #Calculate the partial traspose over the
↪  2nd (B) 2x2 system.
```

```
  reshaped_matrix = decomposeProductTensor(mat, [2,2]) #Decompose it
  trMatrix= np.einsum('ijkl->ilkj', reshaped_matrix)
  return trMatrix.reshape([4,4])
```

Una vez realizado esto, el cálculo es directo

```
rhoABC=np.array(rhot0.subs(t0,np.pi)).astype(np.complex128).round(10)
sigmaAB=myPartialTrace4x3in2ndSpace(rhoABC)

#Now, calculate eigenvalues

wRho, vRho = np.linalg.eig(myPartialTraspose4x3in2ndSpace(rhoABC))
wSigma, vSigma = np.linalg.eig(myPartialTraspose2x2in2ndSpace(sigmaAB))

print("Autovalores para rho_ABC^trC: {}".format(wRho))
print("Autovalores para sigma_ABC^TrB: {}".format(wSigma))
```

Los autovalores obtenidos para $\rho_{ABC}^{\mathrm{tr}_C}$ son los siguientes

$$[0.61085, 0.35694, -0.22729, 0.18625, -0.11450, -0.05954, 0.11005, 0.080107, 0.05374, 0.00338, 0, 0]$$

Los autovalores para σ_{AB} son los siguientes

$$[-0.06056331, 0.59198195, 0.33028734, 0.13829401]$$

con multiplicidad uno cada uno de ellos.

El criterio de Peres-Horodecki es concluyente para respuestas negativas (de la misma forma que lo era el test de Miller-Rabin utilizado en el ejercicio 1, pues ambos usan un argumento de demostración por contrarrecíproco, donde una de las implicaciones no es en general cierta). Como podemos ver, ambos estados tienen autovalores negativos, por lo que ambos están necesariamente entrelazados[4], lo que completa el ejercicio.

References

[Agrawal et al., 2004] Agrawal, M., Kayal, N., and Saxena, N. (2004). Primes is in p. *Annals of mathematics*, pages 781–793.

[Galindo and Martin-Delgado, 2002] Galindo, A. and Martin-Delgado, M. A. (2002). Information and computation: Classical and quantum aspects. *Rev. Mod. Phys.*, 74:347–423.

[Goemans, 2013] Goemans, M. (2013). *Principles of Discrete Applied Mathematics, lecture notes: Huffman Codes*. MIT Mathematics.

[4]Lo que hemos probado para el caso de ρ es que existe entrelazamiento entre el subsistema AB y el subsistema C, lo que prueba también que existe entrelazamiento.

[Horodecki et al., 1996] Horodecki, M., Horodecki, P., and Horodecki, R. (1996). On the necessary and sufficient conditions for separability of mixed quantum states. *Phys. Lett.*, A223:1.

[Knuth, 1997] Knuth, D. E. (1997). *The Art of Computer Programming, Volume 1 (3rd Ed.): Fundamental Algorithms*. Addison Wesley Longman Publishing Co., Inc., USA.

[Krammer, 2016] Krammer, P. (2016). Quantum entanglement: Detection, classification, and quantification.

[Nielsen and Chuang, 2002] Nielsen, M. A. and Chuang, I. (2002). Quantum computation and quantum information.

[Peres, 1996] Peres, A. (1996). Separability criterion for density matrices. *Phys. Rev. Lett.*, 77:1413–1415.

[Preskill, 1998] Preskill, J. (1998). Lecture notes for physics 229: Quantum information and computation.

[Rabin, 1980] Rabin, M. O. (1980). Probabilistic algorithm for testing primality. *Journal of Number Theory*, 12(1):128 – 138.

[Roweis, 2005] Roweis, S. (2005). *CSC310F Lecture 2: Uniquely Decodable and Instantaneous Codes*. New York University.