Class 01

Recall Haskell

Ana Neri ana.neri@quantalab.uminho.pt

DIUM

February 26, 2021

Goals

- Recall Matrix Calculation.
- Recall Haskell basics;

Matrix Calculation [YM08]

- Transpose, Conjugate and Adjoint
- Matrix Multiplication
- Inner Product
- Tensor Product

Transpose, Conjugate and Adjoint

Transpose

$$A^T[j,k] = A[k,j]$$

Conjugate

$$\overline{A}[j,k] = \overline{A[j,k]}$$

Adjoint

$$A^\dagger = (\overline{A}^T) = \overline{(A^T)} \text{ or } A^\dagger[j,k] = \overline{A[k,j]}$$

Transpose, Conjugate and Adjoint

- Transpose is idempotent .... $(A^T)^T = A$
- Transpose respects addition .... $(A+B)^T = A^T + B^T$
- Transpose respects scalar multiplication .... $(c \cdot A)^T = c \cdot A^T$
- conjugate is idempotent .... $\overline{\overline{A}} = A$
- Conjugate respects addition .... $\overline{A+B} = \overline{A} + \overline{B}$
- Conjugate respects scalar multiplication .... $\overline{c \cdot A} = \overline{c} \cdot \overline{A}$
- Adjoint is idempotent .... $(A^\dagger)^\dagger = A$
- Adjoint respects addition .... $(A+B)^\dagger = A^\dagger + B^\dagger$
- Adjoint relates to scalar multiplication .... $(c \cdot A)^\dagger = \overline{c} \cdot A^\dagger$

Matrix Multiplication

Given $A \in \mathbb{C}^{m \times n}$ and $B \in \mathbb{C}^{n \times p}$, $A \cdot B \in \mathbb{C}^{m \times p}$ is defined as :

$$(A \cdot B)[j,k] = \sum_{h=0}^{n-1} (A[j,h] \times B[h,k])$$

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \cdot \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} (a \times e + b \times g) & (a \times f + b \times h) \\ (c \times e + d \times g) & (c \times f + d \times h) \end{bmatrix}$$

Matrix Multiplication

- Matrix multiplication is associative .... $(A \cdot B) \cdot C = A \cdot (B \cdot C)$
- Matrix multiplication has I_n as a unit .... $I_n \cdot A = A = A \cdot I_n$
- Matrix multiplication distributes over addition ...... $A \cdot (B + C) = (A \cdot B) + (A \cdot C)$
- Matrix multiplication respects scalar multiplication $c \cdot (A \cdot B) = (c \cdot A) \cdot B = A \cdot (c \cdot B)$
- Matrix multiplication relates to the transpose .... $(A \cdot B)^T = B^T \cdot A^T$
- Matrix multiplication respects the conjugate .... $\overline{A \cdot B} = \overline{A} \cdot \overline{B}$
- Matrix multiplication relates to the adjoint .... $(A \cdot B)^\dagger = B^\dagger \cdot A^\dagger$

*commutativity is **not** a property of matrix multiplication.*

Tensor Product

$A = \begin{bmatrix} a_{0,0} & a_{0,1} \\ a_{1,0} & a_{1,1} \end{bmatrix}$ and $B = \begin{bmatrix} b_{0,0} & b_{0,1} \\ b_{1,0} & b_{1,1} \end{bmatrix}$

$$A \otimes B = \begin{bmatrix} a_{0,0} \begin{bmatrix} b_{0,0} & b_{0,1} \\ b_{1,0} & b_{1,1} \end{bmatrix} & a_{0,1} \begin{bmatrix} b_{0,0} & b_{0,1} \\ b_{1,0} & b_{1,1} \end{bmatrix} \\ a_{1,0} \begin{bmatrix} b_{0,0} & b_{0,1} \\ b_{1,0} & b_{1,1} \end{bmatrix} & a_{1,1} \begin{bmatrix} b_{0,0} & b_{0,1} \\ b_{1,0} & b_{1,1} \end{bmatrix} \end{bmatrix} =$$

$$\begin{bmatrix} a_{0,0} \times b_{0,0} & a_{0,0} \times b_{0,1} & a_{0,1} \times b_{0,0} & a_{0,1} \times b_{0,1} \\ a_{0,0} \times b_{1,0} & a_{0,0} \times b_{1,1} & a_{0,1} \times b_{1,0} & a_{0,1} \times b_{1,1} \\ a_{1,0} \times b_{0,0} & a_{1,0} \times b_{0,1} & a_{1,1} \times b_{1,0} & a_{1,1} \times b_{0,1} \\ a_{1,0} \times b_{1,0} & a_{1,0} \times b_{1,1} & a_{1,1} \times b_{1,0} & a_{1,1} \times b_{1,1} \end{bmatrix}$$

Haskell [AM15]

- Solutions of problem set 1

1

Write an Haskell function that:
(a) Calculates the perimeter of a circle given its radius.

```
perimeter_circle :: Floating a => a -> a
perimeter_circle r = 2*pi*r
```

(b) Calculates the area of a circle given its radius.

```
area_circle :: Floating a => a -> a
area_circle r = pi*(r**2)
```

How to install:
`http://docs.haskellstack.org/en/stable/README/`

2

Write an Haskell function to calculate the factorial of a number n. Remember that by convention, fac$(0) = 1$.

```
factorial_ :: Int -> Int
factorial_ 0 = 1
factorial_ x = x * factorial_ (x-1)
```

The definition of a **Recursive** function is made in terms of itself by a self-referential expression.

3

Prime numbers can be very useful for many purposes such as cryptography.
(a) Write a function that returns all numbers from 2 *to a given number n.*

```
all_numbers_up_to_n :: Int -> [Int]
all_numbers_up_to_n n = [2..n]
```

(b) Implement a function to eliminate the multiples of a number n from a list.

```
elim_numbers :: Int -> [Int] -> [Int]
elim_numbers n l = [i | i <- l, mod i n /= 0]
```

Anonymous function is a lambda abstractions and may look like x -> \x+2.
List comprehension: generate a list from another list or lists.
Simple example:
`[x^2 | x <- [1..10]]`
`x^2` is the output function.
| splits output and input
`x <- [1..10]` input function
In exercise 3.b. there is a **predicate**.

3

(c) Admit the Erastothenes sieving algorithm: starting by a list of numbers from $2..n$, the algorithm iteratively takes out the multiples of the prime elements. In each iteration the prime element to use always lies in the next position of the element used in the previous iteration.

```
Initial list :  [2,3,4,5,6,7,8,9,10]
Iteration 1:[2,3,5,7,9]//take out the multiples of 2
Iteration 2:[2,3,5,7]// take out the multiples of 3
Iteration 3:[2,3,5,7]// take out the multiples of 5
...
Iteration n:  [2, 3, 5, 7] // in the end only the primes remain
```

3

Implement a function that returns all primes up to a number n.

```
auxprime :: [Int] -> [Int]
auxprime [] = []
auxprime (x:xs) = x : (auxprime (elim_numbers x xs))
prime_list :: Int -> [Int]
prime_list n = list_prime (all_numbers_upto_n n)
```

3

(d) Implement a function that tests the primality of a given number n.

```
is_prime :: Int -> Bool
is_prime n = elem n (prime_list n)
```

(e) Implement a function to factorize a number into prime factors, based on the previous functions. Example: $15 = 3 \times 5$.

```
factorize :: Int -> [Int]
factorize n = [x | x <- (prime_list n), (mod n x == 0)]
```

4

Matrices are an invaluable tool in quantum information. Here we will try to implement some of the primitive operations involving matrices. Consider that the definition for matrices in Haskell is given by lists of lists [[Int]]. For instance, the representation of the Pauli X matrix reads as follows: $\sigma_x = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \rightarrow [[0,1],[1,0]]$ *Implement functions to:*

(a) Calculate the transpose of a matrix A^T

```
transp :: [[a]] -> [[a]]
transp [] = []
transp ([]:_) = []
transp x = (map head x : transp(map tail x))
```

map function applies a function to each item of a list and return the resulting list.

4

```
transp [[1,2],[3,4]]
⇔ (map head [[1,2],[3,4]]) : (transp (map tail [[2],[4]]))
⇔ [1,3] : (transp [[2],[4])
⇔ [1,3] : (map head [[2],[4]]) : (transp (map tail [[2],[4]]))
⇔ [1,3] : [2,4] : (transp [[],[]])
⇔ [1,3] : [2,4] []  because transp ([]:_) = []
⇔ [[1,3],[2,4]]
```

4

(b) Multiply matrices $A \cdot B$

```
mult :: Num a => [[a]] -> [[a]] -> [[a]]
mult m n = [[sum $ zipWith (*) mr nc | nc <- (transp n)] | mr <- m]
```

See [Bas18].
Recall the matrix multiplication definition of slide 6.
Let $(MN)_{ij}$ be the element of $M \cdot N$ located at the i^{th} row and j^{th} column. It can be calculated by $(MN)_{ij} = M_i \cdot N_j^T$, where A_i is the i^{th} row of matrix M and N_j^T the j^{th} column of matrix N.
We only need to work with two vectors MR (row from matrix M) and NC (column of matrix N). IF MR and NC have lenght n then $MR \cdot NC = \sum_{h=1}^{n} MR_h \cdot NC_h$
zipWith applies the $(\cdot)$ to each elements of the input lists that have the same index.
sum computes a sum of all elements in the list.

4

(c) Calculate the tensor product $A \otimes B$

```
tensor_p :: Num a => [[a]] -> [[a]] -> [[a]]
tensor_p x y = [[ a*b | a <- rx, b <- ry ] | rx <- x, ry <- y ]
```

Recall slide 8.
Let x = [[1,2],[3,4]] and y = [[1,0],[0,2]].
For $X \otimes Y_{1,1}$:
rx = [1,2] and ry= [1,0] => a = 1 and b = 1 => the item in $X \otimes Y_{1,1} = 1 \times 1 = 1$
If it helps, you can run the following variation in your prompt to understand the function behaviour better.

```
> let t x y = [[(rx,ry,a,b,a*b) | a<-rx,b<-ry ]|rx<-x,ry<-y]
```

4

(d) Calculate projection matrices $A \otimes A^T$

```
projection :: Num a => [[a]] -> [[a]] -> [[a]]
projection x = tensor_p x $ transp x
```

Recall slide 4.

5

An n-qubit quantum state can be represented by a column vector with 2^n entries. Implement a function that takes a vector representing a 2-qubit state and applies a CNOT operation to it. Note that a CNOT can be represented by a 4×4 matrix:

$$\mathit{CNOT} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

```
gate :: Num a => [[a]] -> [[a]] -> [[a]]
gate g p = mult g p
```

References

Christopher Allen and Julie Moronuki.
Haskell Programming From First Principles.
Gumroad, 2015.

Martijn Bastiaan.
Matrix multiplication with Clash.
Link here, Jul 2018.
[Online; accessed 25. Feb. 2021].

Noson S. Yanofsky and Mirco A. Mannucci.
Quantum Computing for Computer Scientists.
Cambridge University Press, Cambridge, England, UK, Aug 2008.