Università degli Studi di Napoli “Federico II”

Scuola Politecnica e delle Scienze di Base
Area Didattica di Scienze Matematiche Fisiche e Naturali

Dipartimento di Fisica “Ettore Pancini”

Laurea triennale in Fisica

Verso l'applicazione del differential evolution per la mitigazione dell'errore quantistico

Relatore:
Dott.ssa Autilia Vitiello

Candidata:
Noemi Cicala
Matricola N85001400

A.A. 2021/2022

Sommario

Introduzione……3
Capitolo 1 4
1.2 Il qubit……5
1.3 Il gate……9
1.4 Qiskit……13
Capitolo 2 145
2.1 L'algoritmo Differential Evolution (DE): generalità e introduzione……15
2.2 Funzionamento e analisi del DE……17
Capitolo 3 244
3.1 Errori quantistici e possibili soluzioni …… ……24
3.2 Misurazione della mitigazione degli errori quantistici……24
Capitolo 4……33
Conclusioni……36
Fonti……38

Introduzione

Il Quantum Computing, dal 2016, in concomitanza con il lancio del progetto IBM Quantum Experience ha subito una crescita esponenziale nonché un elevatissimo interesse da parte di vari settori della ricerca e dell'industria. Il lancio del suddetto progetto ha contribuito un accesso pubblico più semplice e immediato ai processori quantistici, composti da elementi di calcolo conosciuti come qubit. Sfortunatamente, per loro natura chimica e ingegneristica i qubit sono sensibili al rumore e questo contribuisce al fatto che il calcolo quantistico può essere influenzato da errori.
Presa tal consapevolezza subito sono iniziate le ricerche sullo sviluppo di metodi aventi lo scopo di mitigare l'errore statistico; tra le operazioni più soggette a errori, c'è sicuramente la misura quantistica.
Generalmente, i metodi di mitigazione per la misurazione quantistica si basano tutti sullo stesso principio: calcolare una matrice cosiddetta di mitigazione che possa andare a minimizzare la differenza dei risultati emessi da un processore quantistico da quelli ideali.
Nel presente lavoro di tesi si propone di studiare l'applicabilità di un algoritmo evolutivo conosciuto come Differential Evolution per effettuare l'operazione di mitigazione di risultati ottenuti da circuiti quantistici. L'algoritmo avrà come soluzione la miglior matrice di mitigazione e come funzione di valutazione una basata sulla distanza di Bhattacharyya.

Il lavoro si articola in quattro capitoli. Nel primo capitolo saranno illustrati i concetti base del quantum computing, in particolare si tratterà del qubit, dei gate quantistici e della libreria Qiskit, utilizzata per la programmazione.

Nel secondo capitolo invece verrà analizzato il funzionamento generale del Differential Evolution, applicandolo ad un esempio per sottolinearne maggiormente l'efficacia.

Nel terzo capitolo si affronterà il problema dell'errore quantistico; si partirà illustrandone la natura e successivamente si analizzerà una possibile risoluzione alla sua minimizzazione: attraverso la formalizzazione della matrice di mitigazione.

Nel quarto infine, si proporrà l'applicabilità del DE per la mitigazione dell'errore, soprattutto studiando una possibile e valida candidata di funzione di valutazione dell'algoritmo: quella basata sulla distanza di Bhattacharyya.

Capitolo 1

Negli ultimi anni sono emerse numerose nuove tecnologie informatiche e il calcolo quantistico è probabilmente quella che richiede il cambiamento di paradigma più significativo da parte degli sviluppatori. I computer quantistici sono stati proposti negli anni '80 da Richard Feynman e Yuri Manin.
L'intuizione alla base del calcolo quantistico deriva da quello che è stato spesso considerato uno dei maggiori imbarazzi della fisica: un notevole progresso scientifico che deve confrontarsi con l'impossibilità di modellare anche sistemi semplici. La meccanica quantistica è stata sviluppata tra il 1900 e il 1925 e rimane la pietra miliare su cui si fondano la chimica, la fisica della materia condensata e altre tecnologie, dai chip dei computer all'illuminazione a LED. Nonostante questi successi, anche alcuni dei sistemi più semplici risultano impossibili da modellare con la meccanica quantistica. Questo perché simulare i sistemi di poche decine di particelle interattive richiede una potenza di calcolo che nessun computer convenzionale riuscirà a garantire nemmeno tra migliaia di anni.
Il modo più semplice per interpretare la teoria quantistica consiste nell'affermare che, a livello quantistico, la materia è disponibile in una vasta gamma di configurazioni possibili, note come stati. A differenza della teoria della probabilità classica, queste diverse configurazioni dello stato quantistico, che possono essere potenzialmente osservate, possono interferire tra loro: questa interferenza impedisce l'utilizzo del campionamento statistico per ottenere le configurazioni dello stato quantistico. È invece necessario tenere traccia di ogni possibile configurazione in cui un sistema quantistico potrebbe trovarsi per poter comprendere l'evoluzione quantistica.
Non c'è quindi speranza che i computer convenzionali riescano mai a simulare la propria dinamica quantistica. Tuttavia, questi sistemi si evolvono rapidamente nel tempo in base alle leggi della meccanica quantistica, beatamente inconsapevoli dell'impossibilità di progettarne e simularne l'evoluzione con la potenza di calcolo convenzionale.
Da qui allore le domande:

- Se la dinamica quantistica è difficile da simulare, cosa accadrebbe se dovessimo creare hardware con effetti quantistici come operazioni fondamentali?
- Sarebbe possibile simulare i sistemi di particelle interattive usando un sistema che sfrutta esattamente le stesse leggi che le governano in natura?

Il nucleo fondamentale del calcolo quantistico consiste nell'archiviazione delle informazioni negli stati quantistici della materia e usa operazioni gate quantistiche per

eseguire calcoli su tali informazioni, sfruttando l'interferenza quantistica e imparando a programmarla.
Un computer quantistico non è un supercomputer che può eseguire tutto più velocemente o che può risolvere qualsiasi problema. I problemi che un computer quantistico può risolvere in modo più efficiente rispetto a un computer classico sono denominati BQP (Polinomiali Quantistici con errori delimitati), ovvero sono risolvibili da un computer quantistico in tempo polinomiale.
Prima di avvicinarsi al linguaggio di programmazione, è consigliabile esaminare i principi di base su cui si fonda il calcolo quantistico. Vedere alcuni concetti importanti del calcolo quantistico, ad esempio il qubit, i gate, la libreria qiskit.

1.1 Il qubit

- Introduzione e differenza con il bit binario unitario
  Come il bit binario è l'unità di informazioni di base in un calcolo classico, un qubit o bit quantistico è l'unità di informazioni di base nel calcolo quantistico.
  Un bit binario classico può rappresentare solo un singolo valore binario, ad esempio 0 o 1, ovvero può trovarsi solo in uno di due stati possibili; invece un qubit, facendo uso di fenomeni meccanici quantistici quali la sovrapposizione (per ottenere una combinazione lineare di due stati), può rappresentare quindi uno 0, un 1 o qualsiasi proporzione di 0 e 1 nella sovrapposizione di entrambi gli stati, con una determinata probabilità per ciascuno di essi.
  Il vantaggio proveniente dalla capacità di sovrapposizione risiede nel fatto che vi è un miglioramento di elaborazione e risoluzione di problemi soprattutto da un punto di vista temporale: infatti la quantità di informazioni che può essere rappresentata da un sistema di qubit cresce in modo esponenziale, anche usando poche unità di qubit, contrariamente a quanto avviene con i bit tradizionali.
  Un’altra differenza con i bit binari tradizionali ricade sulla composizione chimica con i quali i qubit sono realizzati: mentre i primi sono chip familiari basati su silicio, i secondi possono essere creati da ioni intrappolati, fotoni, atomi artificiali o reali oppure quasiparticelle.
  Proprio in base alla loro architettura bisogna fare attenzione anche al metodo di conservazione con il quale sono trattati (temperature prossime allo zero assoluto).
  La sovrapposizione, inoltre dicevamo consente di utilizzare altri fenomeni meccanici quantistici, quali l'interferenza e l'entanglement [1].

Il vantaggio risiede sempre nella possibilità dell'aumento esponenziale delle possibilità di calcolo nonché la riduzione notevole dei tempi di elaborazione, in seguito alla combinazione di tutti e tre i fenomeni; analizzata l'interferenza passiamo alla breve analisi degli altri due

- Interferenza: gli stati dei qubit possono interferire tra loro perché ogni stato è descritto da un'ampiezza di probabilità (ampiezze delle onde)
- Entanglement: grazie a ciò i qubit sono sempre correlati tra loro per formare un singolo sistema; anche quando sono infinitamente distanti, infatti, la misurazione dello stato di uno dei qubit consente di conoscere lo stato dell'altro, senza che sia necessario misurarlo direttamente.

Come per tutte le tecnologie ancora in fase di sviluppo, anche per i qubit vi sono ancora diverse problematiche: uno dei più significativi è costituito dalla natura fragile dei qubit. Per esempio l'entanglement del sistema di qubit con il rispettivo ambiente, potrebbe disturbare con facilità il sistema e provocare decoerenza.

Un primo tentativo di risoluzione a tal problema, firmato Microsoft, è l'utilizzo dei qubit topologici, che vengono stabilizzati manipolando la struttura e circondandoli con composti chimici che li proteggono da contaminazione esterna.

- Rappresentazione di un qubit

Lo stato di un singolo qubit può essere descritto da un vettore di colonna bidimensionale di norma unitaria, ovvero, la somma dei quadrati dei suoi elementi deve essere uguale ad 1.

Pertanto $\begin{bmatrix}\alpha\\ \beta\end{bmatrix}$ rappresenta uno stato qubit se α e β sono numeri complessi che soddisfano $|\alpha|^2 + |\beta|^2 = 1$.

I vettori di stato quantistico $\begin{bmatrix}1\\ 0\end{bmatrix}$ e $\begin{bmatrix}0\\ 1\end{bmatrix}$ hanno un ruolo speciale. Questi due vettori formano una base per lo spazio vettoriale che descrive lo stato del qubit: ciò significa che qualsiasi vettore di stato quantistico può essere scritto come somma di questi vettori di base.

In particolare, il vettore $\begin{bmatrix}x\\ y\end{bmatrix}$ può essere scritto come $x\begin{bmatrix}1\\ 0\end{bmatrix} + y\begin{bmatrix}0\\ 1\end{bmatrix}$.

Anche se qualsiasi rotazione di questi vettori fungerebbe da base perfettamente valida per il qubit, viene scelto questo in particolare e viene denominato *base computazionale*.

Infatti questi due stati quantistici vengono selezionati per corrispondere ai due stati di un bit classico, ovvero 0 e 1.

La convenzione standard prevede di scegliere, dunque:

$$0 \equiv \begin{bmatrix}1\\0\end{bmatrix} \qquad 1 \equiv \begin{bmatrix}0\\1\end{bmatrix}$$

- Misurazione di un qubit

  Cosa si intende con il concetto di “misurazione di un qubit”? Una misurazione corrisponde all'idea informale di "osservare" un qubit, che comprime immediatamente lo stato quantistico in uno dei due stati classici.

  Quando viene misurato un qubit dato dal vettore di stato quantistico $\begin{bmatrix}\alpha\\\beta\end{bmatrix}$, il risultato 0 si ottiene con probabilità $|\alpha|^2$ e il risultato 1 con probabilità $|\beta|^2$.

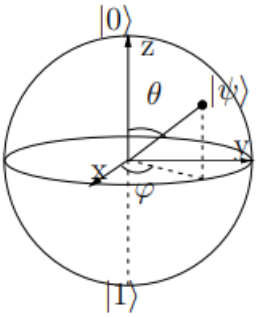
  Pertanto nel risultato 0, il nuovo stato del qubit è $\begin{bmatrix}1\\0\end{bmatrix}$ invece nel risultato 1 lo stato è $\begin{bmatrix}0\\1\end{bmatrix}$.

  Si noti che queste probabilità sommano fino a 1 a causa della condizione di normalizzazione $|\alpha|^2 + |\beta|^2 = 1$; inoltre poiché la probabilità di misurare 0 e 1 dipende dalla grandezza al quadrato dei termini, ne risulta che il segno complessivo del vettore di stato quantistico è irrilevante.

  Un'ultima importante proprietà della misurazione è che non danneggia necessariamente tutti i vettori di stato quantistico: ciò significa che è possibile replicare i dati classici e modificarli in un computer quantistico esattamente come in un computer classico.

- Visualizzazione di qubit

  Una visualizzazione utile di un qubit si può ottenere mediante un’interpretazione geometrica che associa gli stati di un qubit ai punti sulla superficie di una sfera di raggio unitario dove il polo sud della sfera corrisponde a 1 e il polo nord a 0; invece tutte le altre locazioni sono le sovrapposizioni quantistiche di 0 e 1. Questa sfera `e nota come la sfera di Bloch.

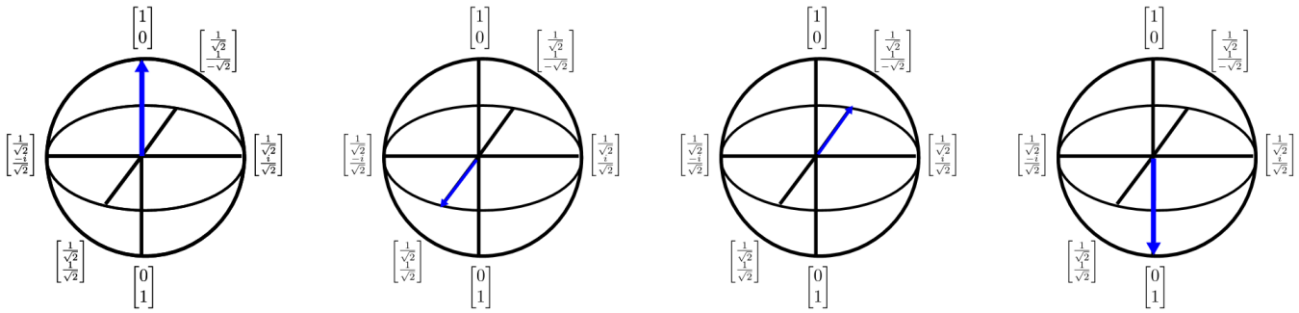
  Esiste una corrispondenza biunivoca tra un generico stato di un qubit:

  $$|\psi> = \alpha\,|0> + \beta\,|1>$$

  e un punto sulla sfera unitaria in R^3 rappresentato come:

  $$cos(\theta/2)|0> + e^{i\phi} sin(\theta/2)|1>$$

dove θ e φ sono numeri reali ossia le coordinate sferiche del punto [2].
L'utilizzo della sfera di Bloch offre un modo intuibile per la descrizione di uno stato quantistico a qubit singolo (che è un vettore complesso bidimensionale) come vettore tridimensionale con valori reali.



Le frecce in questo diagramma mostrano la direzione in cui punta il vettore di stato quantistico e ogni trasformazione della freccia può essere pensata come una rotazione su uno degli assi cardinali [3].

1.2 Il gate

Le operazioni (effettuate tramite i gate quantistici) attraverso cui è possibile manipolare i qubit sono perlopiù un'estensione delle operazioni logiche utilizzate nella computazione classica con il vincolo che la trasformazione di un qubit deve preservare la relazione fra i suoi coefficienti (si parla quindi di trasformazione unitaria).

Formalmente, una trasformazione fra 1-qubit:

$$T\colon \alpha \mid 0\rangle + \beta \mid 1\rangle \longmapsto \alpha' \mid 0\rangle + \beta' \mid 1\rangle$$

deve soddisfare

$|\alpha'|^2 + |\beta'|^2 = 1$..

In modo analogo, vale generalizzazione di tale relazione, ovvero è mantenuta nel caso di *n*-qubit.

Le porte logiche quantistiche, che mimano una particolare operazione sono dunque rappresentate da matrici unitarie.

Formalmente sono operatori unitari che agiscono su un generico qubit nel seguente modo:

$$U\,|\psi> = U[\alpha\,|0> + \beta\,|1>] = \alpha U\,|0> + \beta U\,|1>$$

Il numero di qubit in ingresso in uscita dalla porta deve essere uguale; una porta che agisce su qubit è rappresentata da una matrice unitaria $2^n \times 2^n$.

Gli stati quantistici su cui agiscono le porte sono vettori in 2^n dimensioni complesse.

Le porte quantistiche più comuni operano su spazi a uno o due qubit, proprio come le porte logiche classiche operano su uno o due bit: proprio in riferimento alla possibilità di operare su più di un singolo qubit, è opportuno menzionare il prodotto tensoriale (prodotto di Kronecker) che è usato per combinare stati quantistici; infatti lo stato combinato di due qubit è il prodotto tensoriale dei due qubit.

Il prodotto tensoriale è indicato dal simbolo $\otimes$ e la rappresentazione vettoriale di due qubit è:

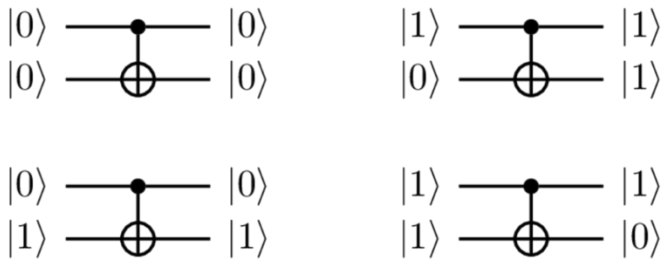
$$|ab\rangle = |a\rangle \otimes |b\rangle = v_{00}|00\rangle + v_{01}|01\rangle + v_{10}|10\rangle + v_{11}|11\rangle \rightarrow \begin{bmatrix} v_{00} \\ v_{01} \\ v_{10} \\ v_{11} \end{bmatrix}$$

Le porte logiche fondamentali sono [4]:

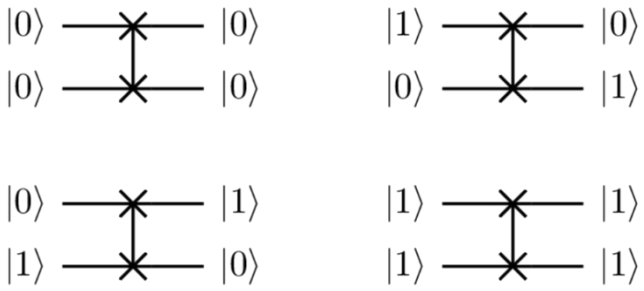
- Il gate *X* o $\oplus$ descrive l'operazione NOT sugli 1-qubit.
  Equivale a una rotazione di π radianti intorno all'asse X della sfera di Bloch. Viene rappresentata dalla prima delle matrici di Pauli: $X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$

- Il gate Y di Pauli agisce su un singolo qubit. Equivale a una rotazione di π radianti attorno all'asse Y della sfera di Bloch. Viene rappresentata dalla seconda delle matrici di Pauli:
$$Y = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}$$

- Il gate $\cdot\!-\!\oplus$ descrive l'operazione CNOT e agisce sullo stato di un 2-qubit come un NOT sulla seconda entrata solo se il valore della prima è 1.



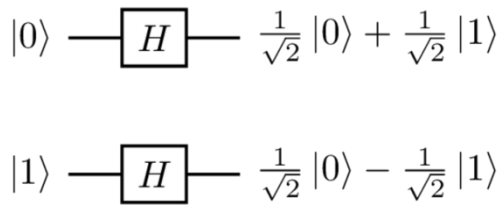
- Il gate $\times\!-\!-\!\times$ descrive l'operazione SWAP e agisce scambiando lo stato di un 2-qubit.



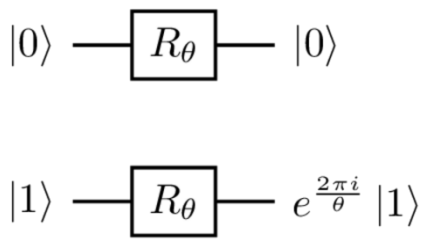
- Il gate Hadamard *H* serve a realizzare il fenomeno della superposizione (uniforme). Infatti, se applicato a |0⟩ oppure |1⟩ restituisce un qubit che può assumere entrambi i valori con equa probabilità. Rappresenta una rotazione

di π intorno all'asse $\frac{(\hat{x}+\hat{z})}{\sqrt{2}}$ nella sfera di Bloch. Equivalentemente, è la combinazione di due rotazioni, di π intorno all'asse Z, e poi di π/2 intorno all'asse Y.

E' rappresentata dalla matrice di Hadamard H: $H = \frac{1}{\sqrt{2}}\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$



- Il gate R_Φ agisce esclusivamente sul coefficiente di |1⟩.
  Si tratta di una famiglia di porte a singolo qubit: è equivalente a tracciare un cerchio orizzontale (una linea di latitudine) sulla sfera di Bloch di ϕ radianti.



La matrice che le rappresenta è: $R_\phi = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\phi} \end{bmatrix}$

dove ϕ è il *phase shift.*

Impostando il phase shift a determinati valori si ottengono le seguenti porte:

1. Il gate Z Pauli (rotazione intorno all'asse Z della sfera di Bloch) $\phi = \pi$ con

$$Z = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\pi} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$$

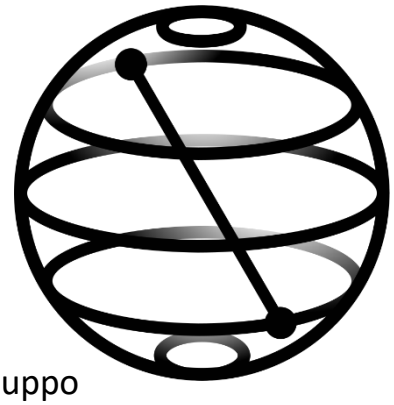
2. Il gate S $\phi = \frac{\pi}{2}$ con

$$S = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\frac{\pi}{2}} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & i \end{bmatrix} = \sqrt{Z}$$

3. Il gate T con $\phi = \frac{\pi}{4}$

$$T = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\frac{\pi}{4}} \end{bmatrix} = \sqrt{S} = \sqrt[4]{Z}$$

1.3 Qiskit



Qiskit è un kit di sviluppo software open source per lavorare con computer quantistici a livello di circuiti, impulsi e algoritmi. Fornisce strumenti per creare e manipolare programmi quantistici ed eseguirli su dispositivi quantistici prototipo su IBM Quantum Experience o su simulatori su un computer locale.
Qiskit è stata fondata da IBM Research per consentire lo sviluppo di software per il loro servizio di cloud computing quantistico, IBM Quantum Experience.
La versione principale di Qiskit utilizza il linguaggio di programmazione Python.
L'obiettivo centrale di Qiskit è creare uno stack software che renda facile a chiunque utilizzare i computer quantistici, indipendentemente dal livello di abilità o dall'area di interesse; inoltre offre la possibilità di sviluppare software quantistico sia a livello di codice macchina di Open QASM, sia a livelli astratti adatti agli utenti finali senza competenze di calcolo quantistico. Questa funzionalità è fornita dai seguenti componenti distinti.

- Qiskit Terra
  L'elemento Terra è la base su cui è costruito il kit, infatti fornisce strumenti per creare circuiti quantistici .
  Permette di costruire esplicitamente i processi che girano su hardware quantistico in termini di porte quantistiche.

Osserviamo l'utilizzo del componente principale di Qiskit per la semplice creazione di uno stato di Bell, utilizzando un circuito a due qubit ai quali vengono applicate fate (CNOT e Hadamard); terminiamo il codice con misurazioni quantistiche, che estraggono un bit da ogni qubit [5].

```
from qiskit import QuantumCircuit

qc=QuantumCircuit(2, 2)

qc.h (0)
qc.cx (0, 1)
qc.measure([ 0 , 1 ],  [ 0 , 1 ])
qc.draw('mpl')
```

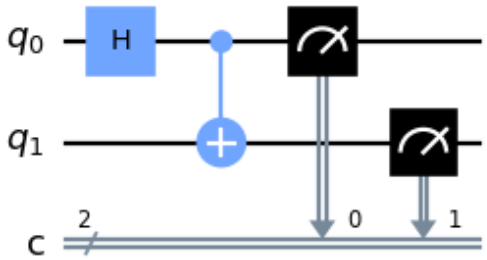


Figura ripresa dal tutorial di Qiskit [5]

- Qiskit Aer
  L'elemento Aer fornisce simulatori di calcolo quantistico ad alte prestazioni con modelli di rumore realistici.
  I simulatori possono anche simulare gli effetti del rumore per modelli di rumore semplici e sofisticati.

Considerando il codice precedente, una volta che il circuito quantistico è stato creato, può essere eseguito su un back-end (hardware quantistico o simulatore).

```
from qiskit  import  Aer

back-end =Aer.get_backend("qasm_simulator")
job =execute(qc, backend)
result =job.result()
print(result.get_counts(qc))
```

- Qiskit Nature
  Qiskit Nature è un framework open source che supporta problemi tra cui calcoli di energia dello stato fondamentale, stati eccitati e momenti di dipolo di molecole, sia a guscio aperto che chiuso. Questi dati di output da un driver possono quindi essere utilizzati come input in Qiskit Nature che contiene la logica in grado di tradurli in una forma adatta agli algoritmi quantistici.

- Apprendimento automatico Qiskit
  Il pacchetto Machine Learning contiene semplicemente set di dati di esempio al momento. Ha alcuni algoritmi di classificazione come QSVM e VQC (Variational Quantum Classifier).

- Ottimizzazione Qiskit
  Qiskit Optimization è un framework open source che copre l'intera gamma dalla modellazione di alto livello dei problemi di ottimizzazione , con conversione automatica dei problemi in diverse rappresentazioni richieste, a una suite di algoritmi di ottimizzazione quantistica di facile utilizzo.

Capitolo 2

2.1 L’algoritmo Differential Evolution (DE): generalità e introduzione

Questo algoritmo, inventato da R. Storn e K. Price nel 1997, è un algoritmo molto efficace per l'ottimizzazione della black-box, chiamata anche ottimizzazione senza derivati.
L'ottimizzazione consiste nella ricerca del minimo di una funzione, dove non conosciamo la sua forma analitica, pertanto nessuna sua derivata può essere calcolata per minimizzarla. Questo tipo di ottimizzazione è molto comune nei problemi del mondo reale, dove la funzione da ottimizzare è molto complessa: per questo motivo il DE è molto popolare per risolvere problemi in molti campi diversi, tra cui astronomia, chimica, biologia, ingegneria e molti altri.
Una caratteristica da non sottovalutare del DE non è solo la sua potenza ma anche la sua semplicità, dato che può essere implementato in poche righe, il che riveste un ruolo fondamentale nell’ambito della programmazione.

Ecco il codice per l'algoritmo DE che utilizza lo schema *rand/1/bin* [6]:

```
import numpy as np

def de(fobj, bounds, mut=0.8, crossp=0.7, popsize=20, its=1000):
    dimensions = len(bounds)
    pop = np.random.rand(popsize, dimensions)
    min_b, max_b = np.asarray(bounds).T
    diff = np.fabs(min_b - max_b)
    pop_denorm = min_b + pop * diff
    fitness = np.asarray([fobj(ind) for ind in pop_denorm])
    best_idx = np.argmin(fitness)
    best = pop_denorm[best_idx]
    for i in range(its):
        for j in range(popsize):
            idxs = [idx for idx in range(popsize) if idx != j]
            a, b, c = pop[np.random.choice(idxs, 3, replace = False)]
            mutant = np.clip(a + mut * (b - c), 0, 1)
            cross_points = np.random.rand(dimensions) < crossp
            if not np.any(cross_points):
                cross_points[np.random.randint(0, dimensions)] = True
```

```
            trial = np.where(cross_points, mutant, pop[j])
            trial_denorm = min_b + trial * diff
            f = fobj(trial_denorm)
            if f < fitness[j]:
                fitness[j] = f
                pop[j] = trial
                if f < fitness[best_idx]:
                    best_idx = j
                    best = trial_denorm
        yield best, fitness[best_idx]
```

Gli unici due parametri obbligatori che dobbiamo fornire sono *fobj* e *bounds*:

- *fobj:* F(x) funzione da ottimizzare che può definita con *def* o un'espressione lambda.
- *bounds*: un elenco con il limite inferiore e superiore per ciascun parametro della funzione; una semplificazione "programmatica" per differenziare i valori per ciascun parametro, può essere nominare l'elenco con il pedice pari alla variabile al quale quel determinato intervallo deve corrispondere.

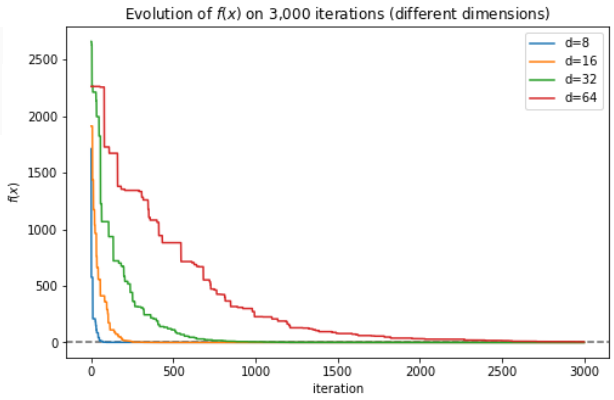
Le prime disparità di questo algoritmo iniziano a sollevarsi cambiando l' ambiente multifunzionale della funzione da ottimizzare: il perché si ritrova nell'effettivo risultato ottenuto in seguito all'utilizzo del di DE, in quanto esso non garantisce di ottenere il minimo globale di una funzione ma in realtà tende ad avvicinarsi al minimo globale gradualmente. Quindi, in generale, più complessa è la funzione, più iterazioni sono necessarie.

Dunque è lecita la domanda: in che modo la dimensionalità di una funzione influisce sulla convergenza dell'algoritmo? In termini generali, la difficoltà di trovare la soluzione ottimale aumenta esponenzialmente con il numero di dimensioni (parametri). Questo effetto è chiamato "maledizione della dimensionalità".

Facciamo un esempio di quanto detto:

se impostiamo la funzione $f(x) = \sum_i^d \frac{x_i^2}{d}$

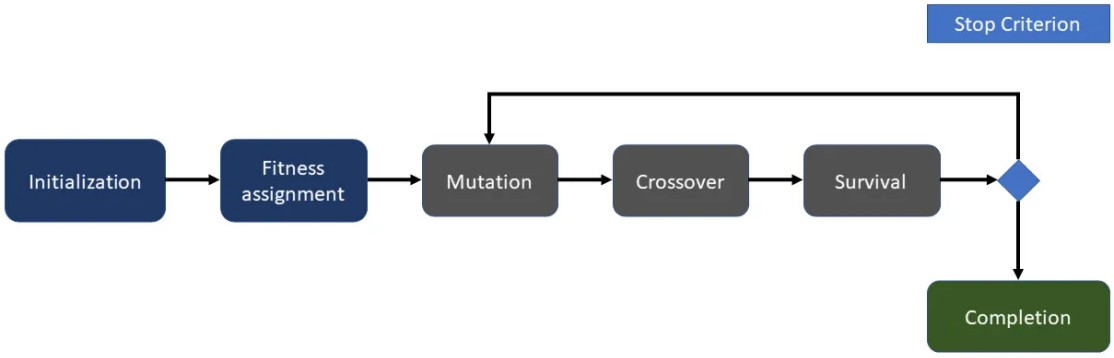
```
for d in [8, 16, 32, 64]:
    it = list(de(lambda x: sum(x**2)/d,
[(-100, 100)] * d, its=3000))
    x, f = zip(*it)    plt.plot(f,
label='d={}'.format(d))
plt.legend()
```



Il grafico chiarisce che quando il numero di dimensioni cresce, cresce anche il numero di iterazioni richieste dall'algoritmo per trovare una buona soluzione.

2.2 Funzionamento e analisi del DE

Il Differential Evolution, come suggerisce il nome, è un tipo di algoritmo evolutivo (EA). Un algoritmo evolutivo è un algoritmo che utilizza meccanismi ispirati alla teoria dell'evoluzione, dove gli individui più adatti di una popolazione (ovvero quelli che hanno i tratti che permettono loro di sopravvivere più a lungo) sono quelli che producono più prole, che a sua volta eredita i tratti migliori dei genitori. Ciò rende più probabile che la nuova generazione sopravviva anche in futuro, e quindi vi è un miglioramento della popolazione nel tempo, generazione dopo generazione. Ciò è possibile grazie a diversi meccanismi presenti in natura, come la mutazione, la ricombinazione e la selezione, tra gli altri.
Gli algoritmi evolutivi applicano alcuni di questi principi per evolvere una soluzione a un problema.
Per quanto detto, quindi, le soluzioni dell'algoritmo sono rappresentate come popolazioni di individui (vettori), dove ogni individuo è rappresentato da un insieme di numeri reali. Questi numeri reali sono i valori dei parametri della funzione che vogliamo minimizzare, e tal funzione "misura" quanto è adatto un individuo.
Le fasi principali dell'algoritmo sono: (1) inizializzazione della popolazione, (2) valutazione, (3) mutazione e ricombinazione, (4) sostituzione.



Analizziamo i passaggi dell'algoritmo con lo scopo di minimizzare la funzione:

$$f(x) = \sum \frac{x_i^2}{n} \; per \; n = 4, dunque \; x = \{x_1, x_2, x_3, x_4\} \; e \; -5 \leq x_i \leq 5$$

Per tanto i parametri sono:

```
fobj = lambda x: sum(x**2)/len(x)
```

```
bounds = [(-5, 5)] * 4
```

1. Inizializzazione della popolazione

ll primo passo in ogni algoritmo evolutivo è la creazione di una popolazione con *popsize*. Un individuo è solo un'istanza dei parametri della funzione fobj. All'inizio, l'algoritmo inizializza gli individui generando valori casuali per ogni parametro all'interno dei limiti dati.
Genero una popolazione di 10 individui aventi 4 parametri ciascuno, per comodità numeri casuali uniformi compresi tra 0 e 1.

```
# Population of 10 individuals, 4 params each (popsize = 10, dimensions = 4)
>>> pop = np.random.rand(popsize, dimensions)
>>> pop

#        x[0]   x[1]   x[2]   x[3]
array([[ 0.09,  0.01,  0.4 ,  0.21],
       [ 0.04,  0.87,  0.52,  0.  ],
       [ 0.96,  0.78,  0.65,  0.17],
       [ 0.22,  0.83,  0.23,  0.84],
       [ 0.8 ,  0.43,  0.06,  0.44],
       [ 0.95,  0.99,  0.93,  0.39],
       [ 0.64,  0.97,  0.82,  0.06],
       [ 0.41,  0.03,  0.89,  0.24],
       [ 0.88,  0.29,  0.15,  0.09],
       [ 0.13,  0.19,  0.17,  0.19]])
```

Applichiamo una trasformazione lineare convertire ogni componente da [0, 1] a [min, max]: questo passaggio è richiesto solo per valutare ogni vettore con la funzione fobj :

```
>>> pop_denorm = min_b + pop * diff
>>> pop_denorm

array([[-4.06, -4.89, -1.  , -2.87],
       [-4.57,  3.69,  0.19, -4.95],
       [ 4.58,  2.78,  1.51, -3.31],
       [-2.83,  3.27, -2.72,  3.43],
       [ 3.  , -0.68, -4.43, -0.57],
       [ 4.47,  4.92,  4.27, -1.05],
       [ 1.36,  4.74,  3.19, -4.37],
       [-0.89, -4.67,  3.85, -2.61],
       [ 3.76, -2.14, -3.53, -4.06],
       [-3.67, -3.14, -3.34, -3.06]])
```

2. Valutazione

A questo punto abbiamo la nostra popolazione iniziale di 10 vettori denormalizzati e quindi possiamo valutarli usando la fobj. Sebbene questi vettori siano casuali dello spazio delle funzioni, alcuni di essi sono migliori di altri, cioè sono inferiori di f(x).

```
>>> for ind in pop_denorm:
>>>    print(ind, fobj(ind))

[-4.06 -4.89 -1.   -2.87] 12.3984504837
[-4.57  3.69  0.19 -4.95] 14.7767132816
[ 4.58  2.78  1.51 -3.31] 10.4889711137
[-2.83  3.27 -2.72  3.43] 9.44800715266
[ 3.   -0.68 -4.43 -0.57] 7.34888318457
[ 4.47  4.92  4.27 -1.05] 15.8691538075
[ 1.36  4.74  3.19 -4.37] 13.4024093959
[-0.89 -4.67  3.85 -2.61] 11.0571791104
[ 3.76 -2.14 -3.53 -4.06] 11.9129095178
[-3.67 -3.14 -3.34 -3.06] 10.9544056745
```

Valutandoli si osserva quindi che il vettore x=`[3.-0.68 -4.43 -0.57]` è il migliore della popolazione, con una F(x) = 7,34.
La valutazione di questa popolazione iniziale viene effettuata e memorizzata nella variabile fitness.

Dettagli sulla Funzione Fitness[7]

La funzione di fitness (nota anche come funzione di valutazione) valuta quanto una data soluzione è vicina alla soluzione ottimale del problema desiderato. Determina quanto è adatta una soluzione.
Per essere una funzione fitness, i requisiti da soddisfare sono i seguenti:

1. La funzione fitness dovrebbe essere chiaramente definita. Il lettore dovrebbe essere in grado di comprendere chiaramente come viene calcolato il punteggio di fitness.
2. La funzione fitness dovrebbe essere implementata in modo efficiente poiché da essa dipende l'efficienza complessiva dell'algoritmo genetico.
3. La funzione di fitness dovrebbe misurare quantitativamente l'idoneità di una data soluzione a risolvere il problema.
4. La funzione fitness dovrebbe generare risultati intuitivi.

La funzione fitness da utilizzare dipende dal problema dato. Trovare una funzione di fitness per il problema dato è la parte più difficile quando si tratta di formulare un problema utilizzando algoritmi genetici.

Non esiste una regola ferrea secondo cui una particolare funzione dovrebbe essere utilizzata in un particolare problema. Tuttavia, alcune funzioni sono state adottate dai data scientist per determinati tipi di problemi.
Tipicamente, per le attività di classificazione in cui viene utilizzato l'apprendimento supervisionato, le misure di errore come la distanza euclidea e la distanza di Manhattan sono state ampiamente utilizzate come funzione di fitness.
Per i problemi di ottimizzazione, le funzioni di base come la somma di un insieme di parametri calcolati relativi al dominio del problema possono essere utilizzate come funzione di fitness.

3. Mutazione e ricombinazione

Come può l'algoritmo trovare una buona soluzione partendo da questo insieme di valori casuali?
Ora, per ogni vettore pop[j] della popolazione (da j=0 a 9), si selezionano altri tre vettori diversi da quello considerato, e li nomimiamo *a*, *b* e *c* .
Partiamo con il primo vettore pop[0] = [-4.06 -4.89 -1. -2.87] detto vettore target e per selezionare *a* , *b* e *c*, si genera una lista con gli indici dei vettori nella popolazione, escluso quello corrente (j= 0):

```
>>> target = pop[j] # (j = 0)
>>> idxs = [idx for idx in range(popsize) if idx != j] # (j = 0)
>>> idxs

[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

poi scelgo a caso 3 indici senza sostituzione con l'apposita funzione:

```
>>> selected = np.random.choice(idxs, 3, replace=False)
>>> selected

array([1, 4, 7])
```

per cui, allora i candidati (dalla popolazione normalizzata) sono:

```
>>> pop[selected]

array([[ 0.04,  0.87,  0.52,  0.  ],  # a
       [ 0.8 ,  0.43,  0.06,  0.44],  # b
       [ 0.41,  0.03,  0.89,  0.24]]) # c

>>> a, b, c = pop[selected]
```

3.1 Ora, creiamo un vettore mutante combinando a, b e c . Come? Gli step sono i seguenti:

- Calcolare la *differenza* (da qui proviene il nome dell'algoritmo) tra b e c;
- Moltiplicare quest'ultima per una costante chiamata fattore di mutazione, detto parametro *mut* (solitamente scelti dall'intervallo [0.5, 2.0]) che ha lo scopo di aumentare il raggio di ricerca (tuttavia al contempo può rallentare la convergenza dell'intero algoritmo);
- Aggiungere, infine, a.

```
# mut = 0.8 di default
>>> mutant = a + mut * (b - c)

array([ 0.35,  1.19, -0.14,  0.17])
```

N.B: si noti che dopo questa operazione, possiamo ritrovarci con un vettore non normalizzato; per rimediare a ciò esistono due metodi comuni: (1) generando un nuovo valore casuale nell'intervallo [0, 1] o (2) ritagliando il numero sull'intervallo, in modo che i valori maggiori di 1 diventino 1 e i valori minori di 0 diventino 0.
Solitamente si sceglie la seconda opzione, per una ragione legata alla brevità del comando (1 riga solo), per cui:

```
>>> np.clip(mutant, 0, 1)

array([ 0.35,  1.  ,  0.  ,  0.17])
```

3.2 Ora il passo successivo da eseguire si chiama ricombinazione.
La ricombinazione consiste nel mescolare le informazioni del mutante con le informazioni del vettore corrente per creare un vettore di prova. Questo viene fatto cambiando i numeri in alcune posizioni nel vettore corrente con quelli nel vettore mutante. Per ogni posizione, decidiamo (con una certa probabilità definita da *crossp*) se quel numero sarà sostituito o meno da quello nel mutante alla stessa posizione.
Per generare i punti di crossover, si generano valori casuali uniformi tra [0, 1] e si controlla se sono minori di *crossp*.
Questo metodo è chiamato crossover binomiale poiché il numero di posizioni selezionate segue una distribuzione binomiale.

```
>>> cross_points = np.random.rand(dimensions) < crossp
>>> cross_points

array([False,  True, False,  True], dtype=bool)
```

Quindi abbiamo ottenuto due True nelle posizioni 1 e 3, il che significa che i valori nelle posizioni 1 e 3 del vettore corrente saranno presi dal mutante tramite la funzione numpy `where`:

```
>>> trial = np.where(cross_points, mutant, pop[j]) # j = 0
>>> trial

array([ 0.09,  1.  ,  0.4 ,  0.17])
```

4. Sostituzione

Dopo aver generato il nostro nuovo vettore di prova,lo si denormalizza e lo si valuta: è questo mutante effettivamente, migliore del vettore corrente (pop[0])? In caso affermativo avviene la sostituzione.

```
>>> trial_denorm = min_b + trial * diff
>>> trial_denorm

array([-4.1,  5. , -1. , -3.3])
```

E ora possiamo valutare questo nuovo vettore con fobj:

```
>>> fobj(mutant_denorm)

13.425000000000001
```

Ma il vettore di prova è peggiore del vettore di destinazione (13,425 > 12,398), quindi il vettore di destinazione viene preservato mentre quello di prova scartato.
Tutti questi passaggi devono essere, ovviamente ripetuti di nuovo per gli individui rimanenti (pop[j] per j=1 fino a j=9), che completa la prima iterazione dell'algoritmo. Dopo questo processo, alcuni dei vettori originali della popolazione saranno sostituiti da altri migliori e, dopo molte iterazioni, l'intera popolazione alla fine convergerà verso la soluzione il che era il nostro scopo iniziale.

Variazioni per il DE
In riferimento al problema di ottimizzazione possono essere necessarie delle piccole modifiche all'interno del corpo dell'algoritmo del DE.
Per esempio lo schema utilizzato precedentemente nell'algoritmo è chiamato rand/1/bin perché

- RAND: i vettori sono scelti casualmente
- 1: abbiamo utilizzato solo 1 vettore di differenza
- BIN: la strategia di crossover utilizzata per mescolare le informazioni della prova e i vettori target era una incrocio binomiale.

Tuttavia ci sono altre varianti, come:
-Per gli schemi di Mutazione

1. Rand/1: $x_{mut}=x_{r1}+F(x_{r2}-x_{r3})$
2. Rand/2: $x_{mut}=x_{r1}+F(x_{r2}-x_{r3}+x_{r4}-x_{r5})$

3. Best/1: $x_{mut}=x_{best}+F(x_{r2}-x_{r3})$
4. Best/2: $x_{mut}=x_{best}+F(x_{r2}-x_{r3}+x_{r4}-x_{r5})$
5. Rand-to-best/1: $x_{mut}=x_{r1}+F_1(x_{r2}-x_{r3}) + F_2(x_{best}-x_{r1})$

-Per gli schemi di Crossover

1. Binomiale (bin): crossover dovuto a esperimenti binomiali indipendenti. Ogni componente del vettore target ha una probabilità p di essere modificata dalla componente del vettore mutante.
2. Esponenziale (exp): è un operatore di crossover a due punti, in cui due posizioni del vettore vengono scelte casualmente in modo che n numeri consecutivi del vettore (tra le due posizioni) siano presi dal vettore mutante.

Capitolo 3

3.1 Errori quantistici e possibili soluzioni

Si può banalizzare l'espressione "effetto del rumore" con il concetto che le uscite ottenute non sono del tutto corrette, ovvero lontane da quelle reali (ideali). L'effetto del rumore, che si verifica durante un calcolo, è in generale assai complesso, poiché si dovrebbe considerare come ogni gate, e dunque le operazione da esse simulate, va incidere sul risultato e che errore va a sviluppare.

Per affrontare la problematica dell'errore quantistico, possiamo far riferimento alla distinzione tra qubit fisico e qubit logico [8].
All'interno di un processore quantistico, ciascun qubit fisico è in corrispondenza uno ad uno con lo stato quantistico di una particella. Abbiamo detto che esistono varie tipologie di qubit fisico a seconda della tecnologia di codifica e più volte abbiamo sottolineato la sua natura instabile. Per correggere gli errori quantistici, più qubit fisici vengono fatti interagire con l'obbiettivo di mantenere stabile nel tempo un unico stato quantistico su cui sia possibile agire in modo non distruttivo: questo insieme di qubit fisici, avente questa caratteristica, viene definito qubit logico.
Allora il qubit logico è un'entità teorica corrispondente alla reale unità di informazione quantistica che un processore quantistico è in grado di controllare ed il numero di qubit logici ne identifica l'effettiva capacità di calcolo associata. Sfortunatamente, il dualismo qubit fisico-logico è un tema ancora poco chiaro: la questione fondamentale che si sta cercando di affrontare è la stima precisa del numero di qubit fisici necessari a realizzarne uno logico (problematica legata alla dipendenza di essa da molteplici parametri). Inoltre, i processori quantistici attuali non dispongono di numerosi qubit fisici, per cui, l'applicabilità di tali tecniche di correzione degli errori quantistici non è applicabile attualmente.

3.2 Mitigazione degli errori quantistici di misurazione

Per iniziare ad entrare nella questione e cercare di sviluppare una procedura per la minimizzazione di tal errore/rumore prodotto iniziamo con la forma più semplice di rumore: quella che si verifica durante la misurazione finale.
In questo caso l'unica procedura da fare è estrarre una stringa di bit come output; quindi per una misura finale di n qubit, questo significa estrarre una delle 2^n possibili stringhe di n bit.
Come semplice modello, possiamo immaginare che la misurazione prima selezioni una di queste uscite in modo perfetto e silenzioso, e quindi il rumore successivamente

faccia sì che questa uscita perfetta venga perturbata casualmente prima che venga restituita come output visibile all'utente.
Date queste impostazioni, è molto facile determinare esattamente quali sono gli effetti degli errori di misurazione.
Si crei un semplice modello di rumore, che capovolge casualmente ogni bit in un output con probabilità p [5].

```
from qiskit.providers.aer.noise import NoiseModel
from qiskit.providers.aer.noise.errors import pauli_error,
depolarizing_error

def get_noise(p):

    error_meas = pauli_error([('X',p), ('I', 1-p)])

    noise_model = NoiseModel()
    noise_model.add_all_qubit_quantum_error(error_meas, "measure")

    return noise_model
```

Si consideri che ogni bit viene capovolto 1% del tempo.

```
noise_model = get_noise(0.01)
```

Adesso è possibile testare gli effetti di tal rumore.
In particolare, si consideri un circuito a due qubit e si preparino gli stati|00⟩,|01⟩, |10⟩ e|11⟩.
Senza rumore, questi porterebbero alle uscite definite.
Il numero di campioni prelevati per ciascun circuito sarà shots=10000.
Vediamo cosa succede con il rumore.

```
aer_sim = Aer.get_backend('aer_simulator')
for state in ['00','01','10','11']:
 qc = QuantumCircuit(2,2)
 if state[0]=='1':
        qc.x(1)
 if state[1]=='1':
        qc.x(0)
 qc.measure([0, 1], [0, 1])
 t_qc = transpile(qc, aer_sim)
 qobj = assemble(t_qc)
 counts=aer_sim.run(qobj,noise_model=noise_model,shots=10000).result()
.get_counts()
 print(state+' becomes', counts)

00 becomes {'01': 102, '11': 3, '10': 97, '00': 9798}
01 becomes {'00': 92, '11': 102, '01': 9806}
10 becomes {'01': 5, '00': 99, '11': 106, '10': 9790}
```

```
11 becomes {'01': 99, '10': 97, '11': 9804}
```

Si osserva quindi che l'output corretto è certamente il più dominante; quelli che differiscono su un solo bit si verificano intorno a 1% del tempo; infine quelli che differiscono su due bit si verificano solo una manciata di volte in 10000 campioni, se non del tutto.

Allora allo stesso modo che dire se eseguissimo un circuito con lo stesso modello di rumore e ottenessimo un risultato come il seguente?

{'10': 98, '11': 4884, '01': 111, '00': 4907}

Qui si verificano '01' '10' per circa 1% di tutti i campioni.
Sappiamo dalla nostra analisi degli stati di base che un tale risultato può essere atteso quando questi risultati non dovrebbero in realtà mai verificarsi (esempio precedente), ma invece il risultato dovrebbe essere qualcosa che differisce da loro solo di un bit.
Possiamo quindi concludere che lo stato iniziale non era semplicemente|00⟩ o |11⟩, ma una sovrapposizione uguale dei due.

In questo esempio *si sono esaminati prima i risultati per ciascuno degli stati di base definiti e sono stati usati dopo per mitigare gli effetti degli errori per una forma più generale di stato.*
Questo è il principio alla base della mitigazione degli errori di misurazione.

Il modo per eseguire la mitigazione algoritmicamente più semplice è usando il supporto matriciale.
Per questo abbiamo bisogno di riscrivere i nostri “dizionari” di conteggio come vettori colonna.
Ad esempio considerato tal conteggio {'10': 96, '11': 1, '01': 95, '00': 9808}, lo si scriverà come:

$$C = \begin{pmatrix} 9808 \\ 95 \\ 96 \\ 1 \end{pmatrix}$$

Le informazioni raccolte dagli stati di base |00⟩, |01⟩, |10⟩ e |11⟩ può quindi essere utilizzato per definire una matrice, che ruota da un insieme ideale di conteggi a uno influenzato dal rumore di misura.
Quindi utilizzando i dati affetti da rumore precedenti si ottiene una matrice:

$$M = \begin{pmatrix} 0.9808 & 0.0107 & 0.0095 & 0.0001 \\ 0.0095 & 0.9788 & 0.0001 & 0.0107 \\ 0.0096 & 0.0002 & 0.9814 & 0.0087 \\ 0.0001 & 0.0103 & 0.0090 & 0.9805 \end{pmatrix}$$

La relazione allora tra il vettore di stato “perfetto” e quello affetto da errore ha proprio come proporzionalità tal matrice, ovvero

$$C_{noise} = M \cdot C_{ideal}$$

Esempio Stato di Bell (|00⟩+|11⟩)/2:

$$\begin{pmatrix} 0.9808 & 0.0107 & 0.0095 & 0.0001 \\ 0.0095 & 0.9788 & 0.0001 & 0.0107 \\ 0.0096 & 0.0002 & 0.9814 & 0.0087 \\ 0.0001 & 0.0103 & 0.0090 & 0.9805 \end{pmatrix} \begin{pmatrix} 5000 \\ 0 \\ 0 \\ 5000 \end{pmatrix} = \begin{pmatrix} 4904.5 \\ 101 \\ 91.5 \\ 4903 \end{pmatrix}$$

Purtroppo, questo è l'esatto contrario di ciò che è necessario.
Invece di un modo per trasformare i dati di conteggio ideali in dati rumorosi, si ha bisogno di un modo per trasformare i dati rumorosi in dati ideali.
In algebra lineare, per una matrice M, la sua matrice inversa M^{-1}:

$$C_{ideal} = M^{-1} \cdot C_{noise}$$

dove

$$M^{-1} = \begin{bmatrix} 1.01978 & -0.01115 & -0.00987 & 0.00011 \\ -0.0099 & 1.02188 & 0.00009 & -0.01115 \\ -0.00997 & 0 & 1.01913 & -0.00904 \\ 0.00009 & -0.01073 & -0.00935 & 1.02009 \end{bmatrix}$$

Applicando questo inverso a C_{noise}, si ottiene un'approssimazione dei conteggi reali.

```
Cmitigated = np.dot(Minv, Cnoisy)
```

Naturalmente, i conteggi dovrebbero essere interi e quindi questi valori devono essere arrotondati.

$$C_{mitigated} = \begin{pmatrix} 5000 \\ 0 \\ 0 \\ 5000 \end{pmatrix}$$

Metabolizzata l'idea che vi è dietro tal procedimento, andiamola ad applicare usando Qiskit: questo gestisce la raccolta dei dati per gli stati di base, la costruzione delle matrici e il calcolo dell'inverso; quest'ultimo in modo particolare viene effettuato tramite il raccordo a minimi quadrati, che rende il risultato più sofisticato da quello ottenibile dall'uso dello pseudo inverso.

```
from qiskit.ignis.mitigation.measurement import complete_meas_cal,
CompleteMeasFitter
```

Ad esempio, limitiamoci a mitigare gli errori per una coppia di qubit. Per questo definiamo un registro quantistico a due qubit e lo inseriamo nella funzione .complete_meas_cal

```
qr = QuantumRegister(2)
meas_calibs, state_labels = complete_meas_cal(qr=qr, circlabel='mcal')
for circuit in meas_calibs:
    print('Circuit',circuit.name)
    print(circuit)
    print()
```

si visualizzeranno, pertanto i seguenti stati

```
Circuit mcalcal_00 → |00>
        ░ ┌─┐
q40_0: ─░─┤M├──
        ░ └╥┘┌─┐
q40_1: ─░──╫─┤M├
        ░  ║ └╥┘
 c0: 2/═══╩══╩═
          0  1
```

```
Circuit mcalcal_10 → |10>
             ░ ┌─┐
q40_0: ──────░─┤M├──
       ┌───┐ ░ └╥┘┌─┐
q40_1: ┤ X ├─░──╫─┤M├
       └───┘ ░  ║ └╥┘
 c0: 2/═══════╩══╩═
              0  1
```

```
Circuit mcalcal_01 → |01>
       ┌───┐ ░ ┌─┐
q40_0: ┤ X ├─░─┤M├──
       └───┘ ░ └╥┘┌─┐
q40_1: ──────░──╫─┤M├
             ░  ║ └╥┘
 c0: 2/══════╩══╩═
             0  1
```

```
Circuit mcalcal_11 → |11>
       ┌───┐ ░ ┌─┐
q40_0: ┤ X ├─░─┤M├──
       ├───┤ ░ └╥┘┌─┐
q40_1: ┤ X ├─░──╫─┤M├
       └───┘ ░  ║ └╥┘
 c0: 2/═══════╩══╩═
              0  1
```

-1: senza rumore

```
t_qc = transpile(meas_calibs, aer_sim)
qobj = assemble(t_qc, shots=10000)
cal_results = aer_sim.run(qobj, shots=10000).result()
```

con i risultati, allora si costruisce la matrice M:

```
meas_fitter = CompleteMeasFitter(cal_results, state_labels,
circlabel='mcal')
print(meas_fitter)
```

$$M= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Ma come ci si aspetta nel caso senza rumore la matrice di calibrazione è pari alla matrice identità.

-2: con rumore, cosa accade se gli errori sono dieci volte più probabili di prima(10%):

```
noise_model = get_noise(0.1)
t_qc = transpile(meas_calibs, aer_sim)
qobj = assemble(t_qc, shots=10000)
cal_results = aer_sim.run(qobj, noise_model=noise_model,
shots=10000).result()

meas_fitter = CompleteMeasFitter(cal_results, state_labels,
circlabel='mcal')
print(meas_fitter)
```

$$M= \begin{bmatrix} 0.806 & 0.0873 & 0.0938 & 0.0107 \\ 0.0957 & 0.8103 & 0.0102 & 0.0893 \\ 0.0874 & 0.0108 & 0.8028 & 0.0852 \\ 0.0109 & 0.0916 & 0.0932 & 0.8148 \end{bmatrix}$$

Quanto e come si può mitigare questo rumore?
Si consideri nuovamente lo stato di Bell (|00⟩+|11⟩)/√2:

```
qc = QuantumCircuit(2,2)
qc.h(0)
qc.cx(0,1)
qc.measure([0, 1], [0, 1])
```

```
t_qc = transpile(qc, aer_sim)
qobj = assemble(t_qc, shots=10000)
results = aer_sim.run(qobj, noise_model=noise_model, shots=10000).result()
noisy_counts = results.get_counts()
print(noisy_counts)
```

adesso si visualizzerà:

```
{'01': 927, '10': 916, '11': 4069, '00': 4088}
```

In Qiskit si può ridurre il rumore creando un oggetto filtro di misurazione

```
meas_filter = meas_fitter.filter

mitigated_results = meas_filter.apply(results)
mitigated_counts = mitigated_results.get_counts()
```

Per vedere i risultati più chiaramente, tracciamo sia i risultati rumorosi che quelli mitigati

```
from qiskit.visualization import plot_histogram
noisy_counts = results.get_counts()
plot_histogram([noisy_counts, mitigated_counts], legend=['noisy',
'mitigated'])
```

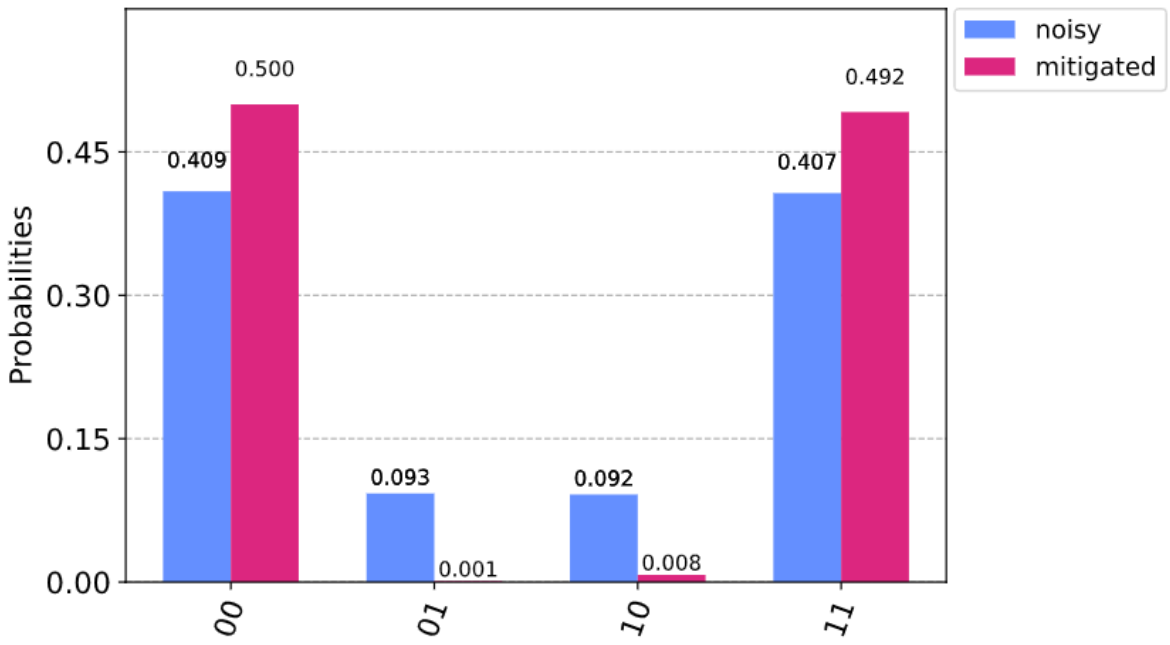


Figura ripresa dal tutorial Qiskit

Si possono osservare dall'istogramma che ora:

- Le probabilità mitigate degli stati |00> e |11>, quali gli unici presenti per la creazione dello stato di Bell, tendono a quelle ideali.
- Le probabilità mitigate degli stati |01> e |10>, come si desiderava, tendono a zero.

Già da questi semplici risultati, si osserva l'efficienza e importanza del metodo di mitigazione attraverso la matrice: non ci resta che indagare sullo sviluppo di un metodo che favorisca la matrice di mitigazione ottimale.

Capitolo 4

La proposta di questo progetto è quella di utilizzare l'algoritmo evolutivo, quale quello del Differential Evolution per ottenere la matrice di mitigazione più adatta, questo perché il DE cerca la soluzione ottimale per il particolare problema, manipolando una popolazione di individui, che sono tutti potenziali candidati di soluzione.
La selezione viene fatta in base alla qualità del particolare individuo della popolazione: analiticamente parlando, questo è il compito affidato alla cosiddetta "fitness function".
Trasportando quanto detto al nostro reale problema di ottimizzazione è semplice capire le seguenti analogie:

- Per popolazione si intende l'insieme di tutte le possibili matrici di mitigazione per l'errore quantistico;
- Per gli individui della popolazione, si intendono le effettive matrici di mitigazione di dimensione 2^nx2^n, dove n è il numero di qubit del registro che si desidera correggere;

La fitness function che meglio propone l'ottimizzazione del problema, dunque che ottimizza la ricerca della migliore matrice tra tutte quelle possibili, si basata sulla distanza di Bhattacharyya[9] ed ha la seguente espressione:

$$F(S) = \frac{1}{2^n} \sum_{i=1}^{2^n} BD(P_i^r, \hat{P}_i^r)$$

dove

- S è la matrice di mitigazione candidata (individuo), appartenente alla popolazione
- P_i^r è il vettore di probabilità ideale per il circuito quantico legato all'i-esimo stato di base.

A causa della natura probabilistica dei computer quantistici, lo stesso circuito quantistico viene eseguito molte volte, pertanto il valore di aspettazione è considerato il più frequente sul numero di volte in cui viene eseguito (solitamente si definisce un parametro 'shots'). Nel caso di un circuito di n qubit allora, il risultato dell'esperimento viene inserito in un vettore C detto 'count', i cui elementi sono il numero di occorrenze per ognuno dei 2^n possibili stati di base. Solitamente però si preferisce utilizzare, invece del vettore di 'count', un vettore P di probabilità:

banalmente i suoi elementi sono quelli del vettore C, divisi per il numero di 'shots' (cioè per quante volte è stato eseguito quel particolare circuito).

- $\hat{P}_i^r$, invece, è il vettore di probabilità rumoroso per il circuito quantico legato all'i-esimo stato di base (ottenuto dopo r shots)

Questa è calcolata come $\hat{P}_i^r = \varphi(S \cdot C_i^r)/r$, ovvero attraverso una funzione usata qualora $\hat{C}_i^r$, cioè il vettore 'count' mitigato, non soddisfacesse delle proprietà:
(1) Tutti gli elementi positivi, (2) la somma di essi è uguale al numero di shots

- BD è la distanza di Bhattacharyya;

In statistica, la distanza di Bhattacharyya è una misura della somiglianza di due distribuzioni di probabilità. È correlato al coefficiente Bhattacharyya, che è una misura statistica della sovrapposizione di due serie di campioni.
Per due distribuzioni di probabilità discrete P=(p_i,….,p_k) e Q=(q_i,…,q_k) definite sullo stesso spazio di probabilità, la distanza di Bhattacharyya è calcolata da:

$$D_B(p,q) = -ln(BC(P,Q)$$

dove:

$$BC(P,Q) = \sum_{i=1}^{k} \sqrt{p_i q_i}$$

è il coefficiente di Bhattacharyya.

A causa della natura probabilistica dei computer quantistici, lo stesso circuito quantistico viene eseguito molte volte, pertanto il valore di aspettazione è considerato il più frequente sul numero di volte in cui viene eseguito (solitamente si definisce un parametro 'shots').
Nel caso di un circuito di n qubit allora, il risultato dell'esperimento viene inserito in un vettore C detto 'count', i cui elementi sono il numero di occorrenze per ognuno dei 2^n possibili stati di base. Solitamente però si preferisce utilizzare, invece del vettore di 'count', un vettore P di probabilità: banalmente i suoi elementi sono quelli del vettore C, divisi per il numero di 'shots' (cioè per quante volte è stato eseguito quel particolare circuito).

Dunque adesso definita e analizzata la funzione di valutazione che meglio individuerebbe l'individuo 'perfetto', cioè la migliore matrice di mitigazione, introduciamo brevemente quali sono i passaggi fondamentali del nostro algoritmo:

1. Generazione casuale dell'insieme di matrici, di dimensione 2^nx2^n
2. Inizializzazione del primo miglior candidato, tramite la fitness function

3. Cicli di ottimizzazione: l'algoritmo si evolve attraverso un insieme di generazioni (per cui sono eseguiti tre passaggi 1) la selezione gi individui casuali, 2) l'applicazione del crossover, 3) l'applicazione della mutazione) fino a quando non vengono soddisfatti alcuni criteri di terminazione, primo fra tutti il raggiungimento di un numero fissato di interazioni (parametro necessario alla chiamata del DE)

Conclusioni

In questo lavoro di tesi, si è provveduto a gettare i primi passi per l'applicabilità dell'algoritmo evolutivo denominato Differential evolution per affrontare il problema della mitigazione dell'errore quantistico di misurazione. L'idea di affrontare tale problema con un algoritmo evolutivo nasce dalla letteratura dove tale problema è stato già risolto con promettenti risultati usando i cosiddetti algoritmi genetici. In generale, per applicare algoritmi evolutivi al suddetto problema, è stato necessario analizzare la definizione di una soluzione al problema e di una funzione che ne definisse la qualità. In particolare, la soluzione al problema è rappresentata da una cosiddetta matrice di mitigazione composta da valori reali. Gli algoritmi genetici non nascono come algoritmi che gestiscono soluzioni a valori reali. Da qui, l'idea di usare il lavoro svolto in letteratura e di sostituire gli algoritmi genetici con un algoritmo che lavora unicamente con soluzioni a valori reali come il Differential Evolution. Nel lavoro di tesi si sono gettati i primi passi studiando l'applicabilità di questo algoritmo pensando che possa avere promettenti risultati. Comunque, per dimostrare sperimentalmente questa visione, nel prossimo futuro, bisognerà valutare la matrice di mitigazione generata dal Differential Evolution nella fase di mitigazione di circuiti creati randomicamente allo scopo di verificare quanto riesca a migliorare i risultati e ad avvicinarli ai risultati ideali che si otterrebbero senza rumore.

Fonti

1. Veronica Cristiano e Francesco Stocco, *Il qubit: introduzione agli algoritmi quantistici*, Dicembre 2017
2. Alessandra Di Pierro, *Appunti di Quantum Computing*
3. https://azure.microsoft.com
4. Projecto*, Gate logici quantistici a QuBit singolo – Programmeremo un vero computer quantistico,* Aprile 12, 2021
5. https://qiskit.org
6. Pablo Rodriguez-Mier, *A tutorial on Differential Evolution with Pytho,* Settembre 5, 2017
7. Vijini Mallawaarachchi, *How to define a Fitness Function in a Genetic Algorithm?*, Luglio 21, 2017
8. Edoardo Signorini e Francesco Stocco*, Il qubit logico e la correzione degli errori quantistici,* Maggio 2018
9. Giovanni Acampora, Michele Grossi and Autilia Vitiello, *Genetic Algorithms based on Bhattacharyya Distance for Quantum Measurement Error Mitigation*, Ottobre, 2021
10. Suguru Endo, Zhenyu Cai, Simon C. Benjamin, and Xiao Yuan, *Hybrid Quantum-Classical Algorithms and Quantum Error Mitigation,* Febbraio 1, 2021
11. Bruno Scalia C. F. Leite, *Differential Evolution: An alternative to nonlinear convex optimization,* Aprile 26, 2022
12. Mariacarla Staffa, *Lectures of Computational Intelligence*, 2020/2021
13. https://www.python.org
14. https://learn.microsoft.com