What could a Quantum PSO be?

Maurice Clerc

What could a Quantum PSO be?

Maurice Clerc*

January 25, 2024

Abstract

The "quantum" Particle Swarm Optimization algorithms that have been published so far are not particularly convincing. There is potential to explore a more quantum approach, albeit in a hybrid manner. The primary challenge lies in the fact that PSO relies on an implicit probability distribution rather than an explicit one. Therefore, it remains uncertain whether we can completely eliminate the use of classical computations, or if at least a few of them are indispensable.

Motivation

If you search using the keywords "quantum particle swarm optimization," you can easily locate numerous papers (Sun, Lai, and Wu 2019;Blackwell and Branke 2004;Sun, Xu, and Feng 2004;Yang, Wang, and jiao 2004;Shuyuan, Min, and Licheng 2004;J. Liu, Xu, and Sun 2005;S. Mikki and Kishk 2005;Oliveira et al. 2006;S. M. Mikki and Kishk 2006;Sun, Fang, et al. 2012;G. Liu et al. 2019 ;Fallahi and Taghadosi 2022; Flori, Oulhadj, and Siarry 2022) to name a few.

Upon careful examination of these papers, it becomes evident that the methods they propose do not truly incorporate concepts such as superposition of states and entanglement, and consequently, they cannot leverage the capabilities of a quantum computer.

For instance, in (Flori, Oulhadj, and Siarry 2022), the proposed QUAPSO method mentions superposition but essentially considers, for each particle, a small user-defined number of velocities to test. The method then retains the velocity that produces the best result. In (Fallahi and Taghadosi 2022) at each time step and for each particle, a random choice is made between two possible next positions.

It is important to note that the critique here does not concern the efficiency of these methods; some of them perform reasonably well. The point is that labeling them as "quantum" may not be entirely appropriate. These algorithms could not, moreover, be executed on a real quantum computer, even in principle.

*Maurice.Clerc@WriteMe.com

Thus, the question arises: Can we define a "more quantum" Particle Swarm Optimization (PSO), particularly for discrete problems? In this context, I am exclusively considering binary problems, as a discrete problem can always be transformed into a binary one.

Let's embark on this exploration by following three key steps:

1. Tackling a straightforward binary problem using a quantum method, aiming to grasp the manipulation of classical entities such as qubits, unitary operators, and states. Feel free to skip this section if you are already familiar with quantum computing.

2. Exploring a classical quantum optimizer, specifically the Grover Adaptive Search, for a comprehensive comparison. Once again, if you are acquainted with this, you can proceed to the next section.

3. Establishing the framework for a pure quantum binary PSO.

About the last part, unfortunately, I haven't had much success with this task so far for the proposed algorithms are hybrid. I'm sharing this research in the hope that someone might find a way to make it better.

Because my laptop is not powerful enough the examples given here are so small that even the exhaustive search is sometimes better than the sophisticated methods to solve them. They are here just to illustrate algorithms that may become really interesting for bigger problems.

Part I
Playing with qubits, gates and states

In this discussion, we focus solely on the pure mathematical approach to quantum methods, excluding any physical interpretation. This is particularly important due to the existence of multiple interpretations, with no consensus on determining the "correct" one.

Technically speaking, a quantum method can be delineated through manipulations of qubits using operators, often referred to as gates, which are essentially unitary matrices[1]. I assume you are familiar with these concepts, but you can refer to the Appendix 6 for a more detailed explanation.

Alternatively, a quantum method can be depicted as a quantum circuit, where a computation involves a series of quantum gates and measurements (D. C. Marinescu and G. M. Marinescu 2012). Notably, the circuit may exhibit dynamic behavior, with certain qubits being reset and reused (Hua et al. 2022).

[1] I assume you know what they are but you may have a look at the Appendix 6

A quantum algorithm operates by altering the components of a set of qubits so that the final measurement yields a binary sequence that is most likely the desired outcome. Generally, it is assumed that all qubits start in the state $|0|$.

The primary challenge lies in executing these modifications "blindly" since observing or measuring a qubit results in its destruction, providing only a binary outcome of 0 or 1.

Binary function, Boolean function
The usual definition of a binary function is a function that can takes only two values: 0 and 1, no matter what are the variables.
For a Boolean function, the variables can take only two values, True and False and the value of the function is defined by logical manipulations of these variables. But for such a function, you can not define something like sin(True).
So, in what follows, a binary function is simply any function defined on $\{0,1\}^D$.

1 A simple example

Let us promptly provide an illustration based on the W states (Wikipedia 2023).

Our objective is to construct all $m \times n$ binary matrices A that contain exactly one '1' in each row.

There are at least three methods to achieve this:

1. A conventional approach utilizing a deterministic (potentially recursive) algorithm. This poses a stimulating exercise that you should attempt.
2. A stochastic method involves minimizing a fitness function, such as $\left|\prod_{i=1}^{m}\left(\sum_{j=1}^{n} A\left(i,j\right)\right)-1\right|$on $\{0,1\}^{mn}$.
3. A quantum approach involves simultaneously generating all these matrices.

Method 1 is deterministic, making it arguably the superior choice. Method 2, on the other hand, may encounter significant challenges in generating all possible solutions.

Now, let's delve into the details of the latter. The diagram labeled 1.1 illustrates the corresponding quantum circuit for the parameters $m = 3$ and $n = 2$. It's worth noting that the standard convention involves numbering the qubits starting from 0. This circuit accurately produces the $2^3 = 8$ solutions, requiring the utilization of three types of unitary operators (commonly referred to as gates when visually represented). The `M` box serves to signify that the qubit is measured, leading to its destruction, and the outcome—either 0 or 1—is recorded as a classical bit.

Result

If we display the state vector (i.e. the superposition of the eight states) just before measure, which is possible only in a simulation on a classical computer, not on an actual quantum device, we find

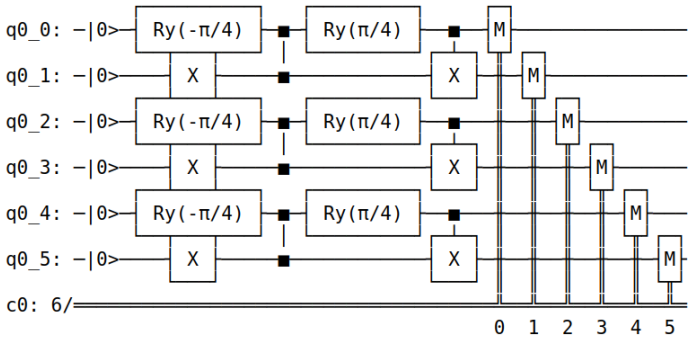


Figure 1.1: Generation of 3x2 binary matrices with one 1 and only one in each row .

$\frac{\sqrt{2}}{4}|010101\rangle+\frac{\sqrt{2}}{4}|010110\rangle+\frac{\sqrt{2}}{4}|011001\rangle+\frac{\sqrt{2}}{4}|011010\rangle+\frac{\sqrt{2}}{4}|100101\rangle+\frac{\sqrt{2}}{4}|100110\rangle+\frac{\sqrt{2}}{4}|101001\rangle+\frac{\sqrt{2}}{4}|101010\rangle$

After measurements one of these eight tensor products will give a binary string. For example $|010101\rangle \Rightarrow 010101$. But only one, with the probability $p = \left(\frac{\sqrt{2}}{4}\right)^2 = \frac{1}{8} = 0.125$. So, if we want several solutions we have to launch several runs (or *shots*). For example, to find the solutions with a probability 0.99999 you need at least $\frac{\ln\left(10^{-5}\right)}{\ln(1-p)} \simeq 86$ shots.

And in practice, of course, we do not find all bit strings with exactly the same probability, as we can see on the figure 1.2 generated after 100 shots.

Note that each bit string must be read from right to left and reformatted as a 3×2 matrix. So, for the second one

$$010110 \Rightarrow \begin{pmatrix} 0 & 1 \\ 1 & 0 \\ 1 & 0 \end{pmatrix}$$

2 Grover Adaptive Search

This is a classical quantum optimizer (Baritompa, Bulger, and Wood 2005; Ortega Ballesteros 2021). We consider here only for further comparison with our quantum PSO on a simple problem which I'll refer to as G6 and defined as follows (see also Figure 2.1 for a possible 1D landscape[2]). There are many local optima but it is relatively easy to escape them for they contains at most two points and even often just one.

Find the binary position $x = (x_1, x_2, x_3, x_4, x_5, x_6)$ of length $n = 6$ that minimizes

[2]There are $2^6 = 64$ possible strings, so 64! such landscapes

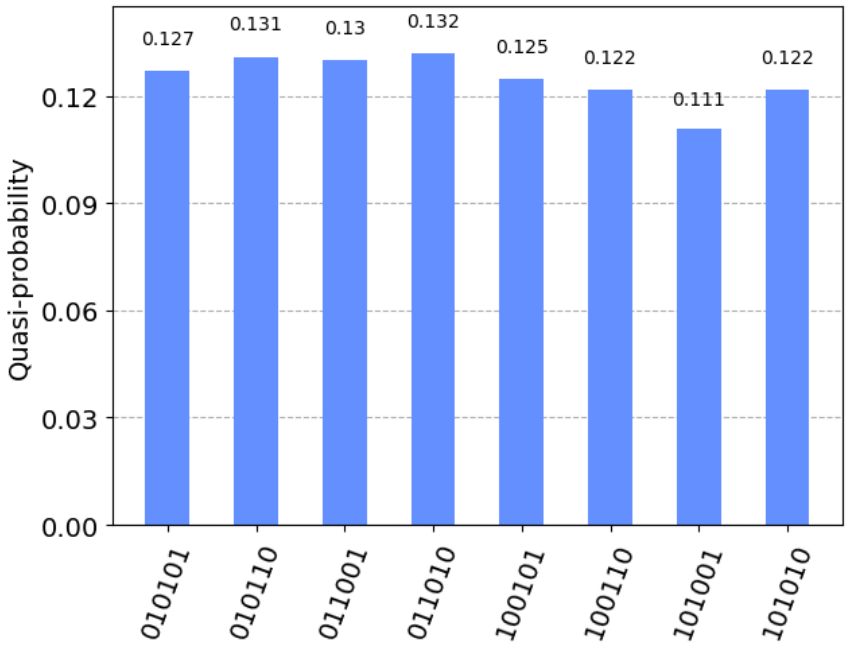


Figure 1.2: After measure the probability of each bit string is not exactly the theoretical one (0.125)

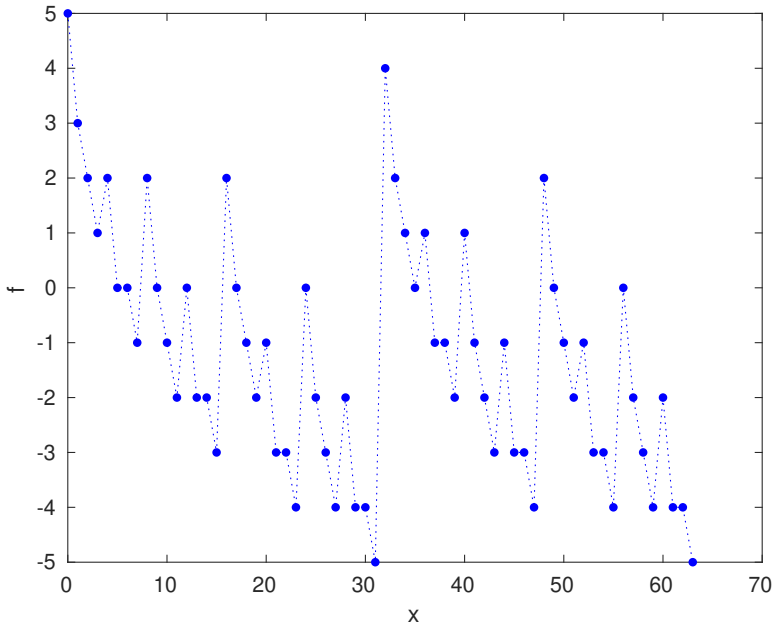


Figure 2.1: G6 problem. A possible landscape on $[0, 64]$. It is multimodal with two minima at -5.

$$f(x) = -15 + \sum_{i=1}^{n-1} (4 - x_i - 2x_{i+1} + x_i x_{i+1}) \tag{2.1}$$

The problem, can be resolved using ten qubits.

It requires six qubits to represent positions, along with m qubits for the possible values of a position (more precisely the difference relatively to a variable threshold). If the maximal value is f_{max} theoretically $m$should be so that

$$f_{max} \leq 2^{m-1} - 1$$

Here $f_{max} = 5$, so we have

$$m = \lceil \log_2(5) + 1 \rceil = 4$$

Let s be the number of solutions. The optimal number of iterations for the Grover search is given by

$$r = \left\lfloor \frac{\pi}{4} \frac{1}{\arcsin\left(\sqrt{\frac{s}{2^{n+m}}}\right)} \right\rfloor$$

As we are not supposed to know there are more than one solution we set $s = 1$ and the formula gives $r = 25$. We will use this value but, actually there are two solutions and 17 would be better, because the efficiency (the probability of success) follows a $\sin^2$ law: too few iterations certainly reduces it but too many may *also* reduce it.

By running the Qiskit code (given in the Appendix 8.1) we indeed find the two solutions, on positions $x = (0, 0, 0, 0, 0, 0)$ and $(0, 1, 1, 1, 1, 1)$ for which the f value is -5 (see the figure 2.2).

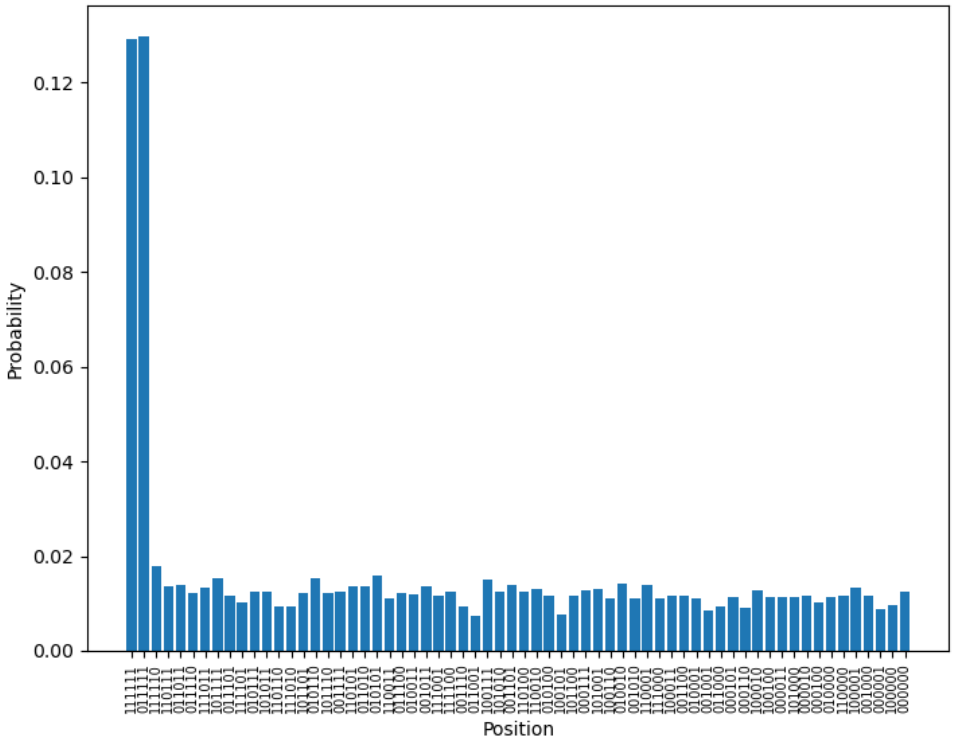


Figure 2.2: Solving a 6-bits problem by Grover's search. Two solutions.

Part II
PSO: from classical to quantum

By combining different strategies it is possible to define many binary PSO variants [Maurice Clerc 2005). But to illustrate the transformation *classical* $\Rightarrow$ *quantum* let us consider just the oldest one [Kennedy and Eberhart 1997). Of course more sophisticated variants do exist (see for example Lee et al. 2008).

3 A classical binary PSO

The problem is defined by a fitness function f on $\{0,1\}^D$ and a position x is said to be better than a position y iff $f(x) < f(y)$.

The velocity update formula is

$$v_{i,(t+1),d} = wv_{i,t,d} + \tilde{c}_{i,t,d}\left(p_{i,t,d} - x_{i,t,d}\right) + \tilde{c}_{i,t,d}\left(g_{i,t,d} - x_{i,t,d}\right) \tag{3.1}$$

and the position update formula[3]

$$\begin{cases} x_{i,(t+1),d} & = 1 - x_{i,t,d} \quad if\left(U\left(0,1\right) < S\left(v_{i,(t+1),d}\right)\right) \\ & \quad 0\, else \end{cases} \tag{3.2}$$

where

w is the inertia weight

i is the rank of the particle in the swarm of size N.

t is the time step.

d is the current dimension.

Note that terms like $p - x$ and $g - x$ can take only three values: -1, 0 and 1.

$\tilde{c}_{i,t,d}$ is a random positive number generated for each (i,d,t), whose upper limit is an undefined parameter. Usually drawn from the uniform distribution.

$x_{i,t} = (x_{i,t,1}, \ldots, x_{i,t,D})$ the "position" of the particle i at time t, an element of $\{0,1\}^D$.

$v_{i,t} = (v_{i,t,1}, \ldots, v_{i,t,D})$ the "velocity" of the particle i at time t, here a value in $]-V_{max}, V_{max}[^D$. Note that at in 1997 the velocity in PSO, including this binary version, was limited by an user-defined parameter

$$v_{i,(t+1),d} \to \max\left(\min\left(v_{i,(t+1),d}, V_{max}\right), -V_{max}\right) \tag{3.3}$$

Only later the constriction coefficient made it possible to avoid this arbitrary parameter (M. Clerc and Kennedy 2002). In Kennedy and Eberhart 1997 V_{max} is set to 6.

$p_{i,t} = (p_{i,t,1}, \ldots, p_{i,t,D})$ the best "position" found so far at time t by the particle i, usually called "previous best".

$g_{i,t} = (g_{i,t,1}, \ldots, g_{i,t,D})$ the best of the best positions found so far at time t by the neighbors of the particle i.

$U\left(0,1\right)$ a random value drawn from the uniform distribution on $[0,1]$.

S a logistic transformation that maps $]-\infty, +\infty[$ to $]0,1[$:

$$S\left(u\right) = \frac{1}{1 + e^{-\lambda u}} \tag{3.4}$$

for which $\lambda = 2$ seems to be a good choice.

[3] In the original paper the formula is wrongly noted $\begin{cases} x_{i,(t+1),d} & = 1\, if\left(U\left(0,1\right) < S\left(v_{i,(t+1),d}\right)\right) \\ & \quad 0\, else \end{cases}$

Note that if you consider bits as logical values you can replace $1 - x_{i,t,d}$ by $\neg x_{i,t,d}$ (i.e. switching the bit).

The idea is that if for a dimension d the velocity is "high" then the d-th bit of the position should probably be switched.

The mapping function S can easily be modified. Not clear whether there exists a "best" one or not, though.

The neighborhood N_i of the particle i is defined by what is called "topology" and there are many possible ones, either constant, variable or adaptive.

At each time step and for each particle i we apply the velocity update and the position update formulae and then we take the fitness function f into account to possibly update the previous best

$$\begin{cases} p_{i,(t+1)} & = & x_{i,(t+1)} \, if \, f\left(x_{i,(t+1)}\right) < f\left(p_{i,t}\right) \\ & = & p_{i,t} \, else \end{cases} \tag{3.5}$$

and the best previous best in the neighborhood

$$g_{i,(t+1)} = \arg\left(\min_{j \in N_i}\left(f\left(p_{j,(t+1)}\right)\right)\right) \tag{3.6}$$

In what follows, for simplicity I use first the so called global topology: for each particle the neighborhood is the whole swarm so there is just one g.

4 Quantum Imitation

We can try to mimic classical PSO with quantum operations. Actually this is a hybrid approach. The main idea is "Replace bits by qubits", but some classical computations are still needed. There are two difficulties:

- we can not precisely know a position;
- apparently we can not write something as simple as $p_i = x_i$ for cloning an unknown quantum state is impossible.

To cope with the first one, we estimate the probability density by launching many shots with measure. The second one is due to an easy to prove theorem (Wootters and Zurek 1982). However we do not really care here for several reasons:

- It is not valid if you know the state, particularly after a forced (re)initialization.
- Similarly to the No Free Lunch theorem, which is valid only on sets of problems that are closed under permutations, a situation that never occurs in practice, this one is valid only for pure quantum states, not mixed ones (see the Appendix for definitions) that usually have to be used if you want to take advantage of the main quantum properties (superposition, entanglement).

- An imperfect cloning (quasi-copy) is always possible (Bužek and Hillery 1996, Mastriani 2022).
- Finally, the no-cloning theorem holds true only when all applicable operators are unitary. When, for instance, a qubit is reset, not only is its state revealed, but the associated operator is no longer unitary. In such instances, it becomes feasible to effectively "copy" a qubit, a process we will apply in the subsequent discussion. Note that, though, it implies that the quantum device can perform qubit resets.

4.1 Velocity and position updates

For each i in $\{1, \ldots, N\}$ we define two lists of qubits that can be called *q-particles*:

$x_i = (x_{i,1}, \ldots, x_{i,D})$ for the positions, i.e. the explorer swarm;

$p_i = (p_{i,1}, \ldots, p_{i,D})$ for the previous best positions, i.e. the memory swarm.

In the formula 3.1 a term like $p_{i,t,d} - x_{i,d}$ can be interpreted as "how much" the qubits $p_{i,d}$ and $x_{i,d}$ are different, and depending on its sign it will implies attraction or repulsion.

The "closeness" between them can be evaluated by the *fidelity* in $[0, 1]$ (see the Appendix 6) and therefore the "distance" by *1-fidelity* which is in $[0, 1]$. But as we want a value that can be either positive or negative we use in fact $1 - 2 \times fidelity$, which is in $[-1, 1]$.

The formula 3.1 becomes

$$v_{i,d} \Leftarrow w v_{i,d} + \tilde{c}_{i,d} \left(1 - 2 \left|\left| p_{i,d} x_{i,d} \right|\right|^2\right) + \tilde{c}_{i,d} \left(1 - 2 \left|\left| g_d x_{i,d} \right|\right|^2\right) \tag{4.1}$$

Then we apply a modified mapping so that the result is in [-1,1]:

$$S(u) = \frac{2}{1 + e^{-\lambda u}} - 1 \tag{4.2}$$

From a quantum point of view the formula 3.2 has to be transformed

$$x_{i,(t+1),d} = R_y \left(\pi S\left(v_{i,d}\right), x_{i,d}\right) \tag{4.3}$$

Why this rotation? Let's consider the extreme case $x_{i,d} = 1 \times |0> + 0 \times |1>$ and $S(v_{i,d})$ (almost) equal to 1. Then the Ry rotation switches the qubit, as in the non-quantum approach (formula 3.2). But here the process is smoother for smaller S values: only the probabilities of $|0>$ and $|1>$ are modified.

Example

We have $x_{i,d} = \sqrt{0.2} \times |0> + \sqrt{0.8} \times |1>$ and a better position has been found for which the qubit d is $\sqrt{0.9} \times |0> + \sqrt{0.1} \times |1>$. So the qubit $x_{i,d}$ should go "towards" this one. Assuming there is no more improvement for eight iterations, a possible evolution of $x_{i,d}$is given in Table 1. As expected the probability of state $|0>$ (and, of course also the one of $|1>$) first tends to the "good" one (0.9) and then oscillates.

Table 1: How a qubit goes "towards" another thanks to R_y rotations. Note the oscillation.

$x_{i,d}$		Probability |0>	Probability |1>
0.447	0.894	0.2	0.8
0.269	0.963	0.072	0.928
-0.035	0.999	0.001	0.999
-0.506	0.862	0.256	0.744
-0.871	0.491	0.759	0.241
-0.947	-0.322	**0.896**	**0.104**
-0.560	-0.829	0.313	0.687
0.033	-0.999	0.001	0.999
0.713	-0.701	0.509	0.491

We can estimate what could be the maximum value for $v_{i,d}$. The worst-case scenario occurs repeatedly when the fidelities are zero and the random $\tilde{c}_{i,d}$ equal to c. At the limit, as $w < 1$ we find

$$v_{max} = \frac{c}{1-w} \tag{4.4}$$

4.2 Global best version

For simplicity we use here the global best topology (although it is bad for classical multimodal problems).

The formula 4.3 becomes

$$v_{i,d} \Leftarrow w v_{i,d} + \tilde{c}_{i,d} \left(1 - ||g_d x_{i,d}||^2\right) \tag{4.5}$$

We consider the qubits independently. Remember that each $x_{i,d}$ can be written

$$x_{i,d} = \alpha|0> + \beta|1>$$

Note that we do not even need to use complex numbers.

A pseudo-code is given in the box 1. To be sure you can reproduce the experiments a Qiskit code for simulation (version 0.44.1) is given in the Appendix 8.3.

On a simulation we could have access to the state vector. However it would be a sort of cheating for it is not possible on a real quantum device. So , instead, I run many times the circuit ("shots") to build a quasi-distribution, which can be seen as an estimation of the unknowable state vector. Of course, it means that the accuracy of this estimation is depending on the number of shots. In the code of the "Tools" there is a function that estimate the number of shots for a desired accuracy (say 0.95), but it can not work as soon as the number of qubit is too big.

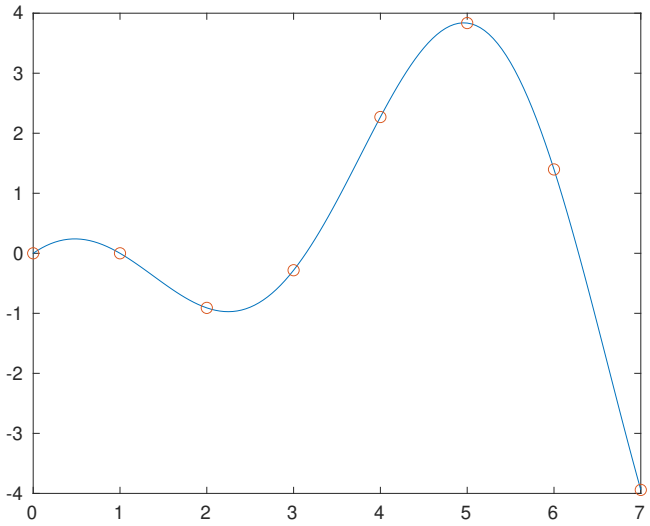


Figure 4.1: A simple bimodal function

Note that in this code the qubits defining the global best position g are destroyed at each iteration, but quickly rebuilt thanks to the other positions that do are rebuilt.

We can test this algorithm with two or three q-particles on a few simple functions.

4.2.1 Sin3 problem

The continuous form of the fitness function is defined on $[0, 7]$ by

$$f(x) = \sin(x)(1 - x) \tag{4.6}$$

It is bimodal as we can see on the figure 4.1.

We convert it into a binary one defined on $\{0, 1\}^3$ thanks to the *binarization technique* (see the Appendix 7)

$$f(b_0, b_1, b_2) = \sin(b_0 + 2b_1 + 4b_2)(1 - b_0 - 2b_1 - 4b_2) \tag{4.7}$$

The solution is $f(1, 1, 1) = -3.942$. Let's try our code with 100 runs and 4 iterations/run. The parameter values are w=0.721 and c=1.193. They are not finely tuned[4] and we keep the same values for the other examples. The results on the table 2 show that, as expected, the success rate increases with the swarm size. And also with the number of shots, but not that much.

[4]These values comes from the classical Standard PSO but nothing proves this is a good choice here.

Algorithm 1 Hybrid Quantum PSO for global best topology

Hybrid Quantum PSO (HQPSO) - Global best topology - Pseudo-code for one run

The circuit is recreated at each loop thanks to values classically saved in the previous one. So the method is hybrid.

```
Swarm size N
Dimension D (i.e. number of bits)
For the N particles i set f_{p,i} = ∞
Loop while no stop criterion is met
    Create a quantum circuit qc:
        N*D qubits for the current positions x_0, ..., x_{N-1}
        D qubits for the global best g
        N*D bits for measure
    If first loop
        Initialize the velocities v at random (they can set to zero)
        Initialize the positions x at random
    else
        Update velocities (see formula 4.5)
            ⇒ v and S
        Update positions (formula 4.3).
            It uses v and S (not a pure quantum approach)
            Can be done thanks to reset and R_y gates
    Measure the positions and
      execute the circuits nshots times
        ⇒ nshots binary strings
    Keep s, the one with the highest probability
    For each position i
        extract from s the corresponding bit string s_i
        evaluate the fitness f(s_i)
        if f(s_i) < f_{p,i}
            x_best = s_i
            f_{p,i} = f(s_i)
            copy x_i to g
```

Table 2: Global best HQPSO - Sin3 problem - 100 runs. The success rate of random search would be 12.5 %.

Swarm size	1	1	2	2	3
Qubits	6	6	9	9	12
Shots	500	500	1024	1024	1024
Iterations/run	4	10	4	10	4
Evaluations	396	1089	382	804	327
Success rate	66 %	92 %	88 %	99 %	98 %

Table 3: Global best HQPSO - G6 problem - 100 runs. The success rate of random search would be 3.125 %.

Swarm size	1	1	2	2	3
Qubits	12	12	18	18	24
Shots	2048	2048	2048	2048	2048
Iterations/run	4	10	4	10	4
Evaluations	493	2131	468	904	597
Success rate	31 %	81 %	84 %	100 %	91 %

During a run a sequence of improvement can be for example $[0.0, 0.0, 1.3970, -3.9419]$.

A circuit (with just two q-particles) is given on the figure 4.2. Note the three CX gates that copy the best position (the first one) to the global best.

4.2.2 G6

4.2.3 Binary Rastrigin

We define the continuous function on $[0, 7]^2$ by

$$f(x_1, x_2) = 20 + (x_1 - 3)^2 - 10\cos(2\pi x_1) + (x_2 - 5)^2 - 10\cos(2\pi x_2) \quad (4.8)$$

This function is highly multimodal as we can see on Figure 4.3a. The minimum is zero on $(3, 5)$.

When considering only integer 2D points it is unimodal (figure 4.3b) but the algorithm doesn't "know" that. Actually it is not really aware of any landscape but we can better understand why the problem is far more difficult[5] than G6 — although the dimension is the same (6) — by looking at a possible 1D landscape (figure 4.3c) whose local optima contain seven points, instead one or two for G6.

The algorithm proposes as a solution the binary string $(101011) = 43$. As Qiskit uses the little-endian notation we can reverse the string and if we split it we indeed get $(110, 101) = (3, 5)$.

[5]Inversely, it's a good example of a problem that becomes simpler when coded in higher dimension.

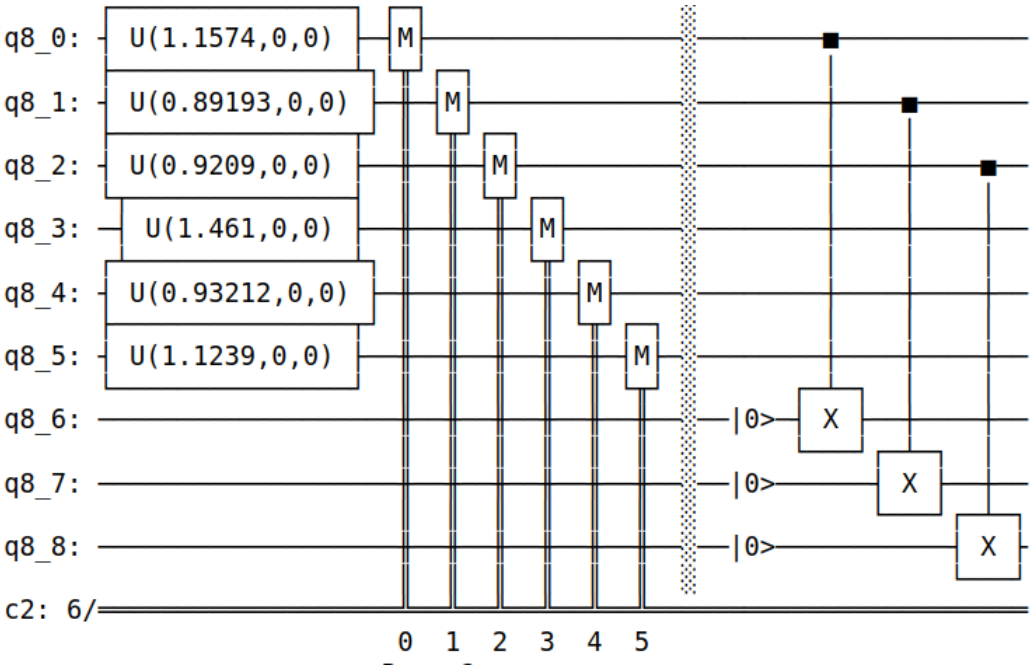
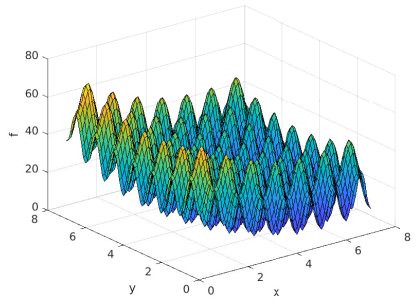


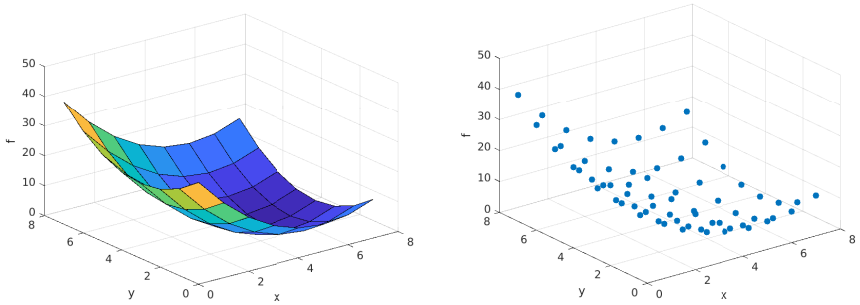
Figure 4.2: A typical circuit. The first q-particle (three qubits) is an improvement and copied to save it.

Table 4: Global best HQPSO - Binary Rastrigin problem on $[0, 7]^2$ - 100 runs. The success rate of random search would be 1.56 %.

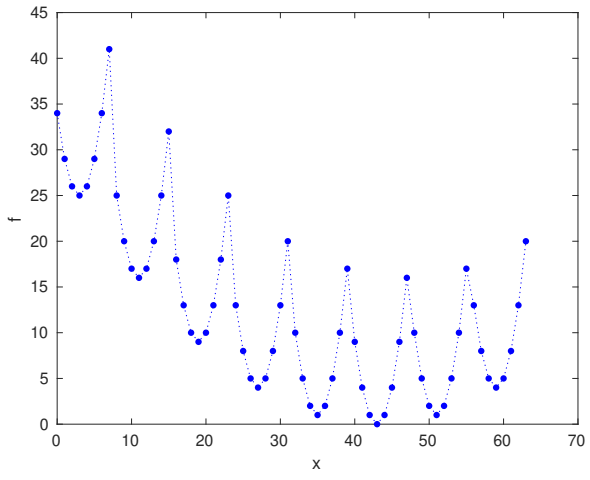
Swarm size	1	1	2	2	2	3	3
Qubits	12	12	18	18	18	24	24
Shots	1024	1024	1024	2048	2048	1024	2048
Iterations/run	4	10	4	4	10	4	4
Evaluations	555	3959	1138	1096	7950	1617	1611
Success rate	10 %	16 %	11 %	13 %	14 %	16 %	15 %



(a) Continuous. Highly multimodal.



(b) On integer 2D points it is unimodal but our HQPSO doesn't "see" that.



(c) In 1D representation on integer points it is multimodal. Each local valley contains seven points so it is difficult to escape.

Figure 4.3: Rastrigin landscapes.

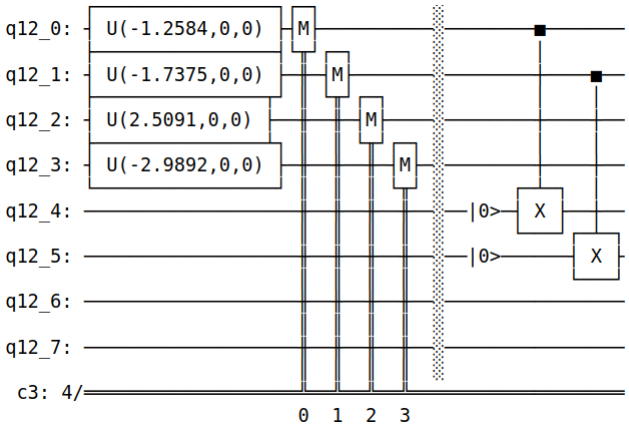


Figure 4.4: A typical circuit ($N = 2$, , K=1,$D = 2$). At this time step explorer 1 (qubits 0 and 1) "moves" and finds a better position. After, thanks to the saved S values, it is rebuilt and copied to its memory (qubits 4 and 5).

Table 5: Local best HQPSO - Some results

	Sin3	Sin3	G6	G6	G6	Rastrigin	Rastrigin
Swarm size	2	2	2	2	3	2	2
Neighborhood size	1	1	1	1	2	1	1
Qubits	24	24	24	24	36	24	24
Shots	1024	1024	1024	1024	2048	2048	2048
Iterations/run	4	10	4	10	4	4	10
Evaluations	594	2440	1046	6936	5904	1112	7798
Success rate	70 %	89 %	21 %	28 %	24 %	10 %	15 %

4.3 Local best topology

We can try with a local best topology, a random variable one with at most K neighbors (like say in SPSO 2007, available on the Particle Swarm Central PSC 2024). The velocity update formula is now the complete one 4.1.

The algorithm is still hybrid and needs more qubits than for the global best topology, ($2ND$ vs $(N + 1)D$) .

The Qiskit code is given in the appendix 8.4. It is a of course longer than the one for the global best approach.

5 Discussion

The above results are not particularly good, even if better than with random search. They are far worse than with exhaustive search, that would need 2^D evaluations, i.e. 64 for $D = 6$. Of course the hope is that it would be different for higher D values.

Although only very simple examples has been treated it seems that, contrarily to the continuous case, the local best topology is not more efficient than the global best one. As it needs less qubits the global best topology seems preferable but, of course, more and bigger examples are needed to conclude. But is it worth?

These two approaches do use qubits and quantum gates (H, CX and R_y) but are not satisfying for at least three reasons:

- Not elegant, for they are hybrid methods.
- They need a lot of qubits. The Grover search needs less qubits. However its Qiskit code used here works well only for Quadratic unconstrained binary optimization (QUBO) but not, for example, on the Rastrigin problem[6].
- They don't truly leverage specific quantum features such as entanglement and superposition. Therefore, in reality, both of them are essentially just another instance of "fake quantum PSO."

For the classical iterative approach let's temporarily call *state* a position in the search space.

The "philosophy" of this approach is then "Progressively find a state better than the previous one".

But for the quantum approach it is "Progressively increase the probability of desirable states".

They appear similar, but they differ, as evident in this study when comparing Grover's approach with our "quantum" PSO ones.

When we assert that a position is represented by a set of qubits, a q-particle, it implies that this q-particle essentially represents all potential positions. Therefore, what we must do is manipulate a probability distribution over the quantum state's space. This implies two observations if we intend to convert a classical iterative algorithm A into a quantum one, qA:

- If A does not explicitly use a probability distribution to sample the next position, it appears extremely difficult to avoid classical computation in qA. Therefore, it is considered hybrid.
- On the contrary, if A is an Estimation of Distribution Algorithm (EDA), defining a purely quantum version might be easier. However, today[7], the published attempts are not very convincing. This is either due to their results not being particularly good or because they, in fact, utilize Grover search. See for example Soloviev, Bielza, and Larrañaga 2021; Morimoto et al. 2023.

[6] For fair comparison I had to transform the function into a quadratic one, in which $cos(x)$ is replaced by $1 - x^2/2$. The results are bad (wrong solutions) with Grover search as with global best HQPSO (two q-particles, 2048 shots and 4 iterations/run) the success rate is about 11 %.

[7] January 2024

So there is still this open question:"Is it worth trying to turn a classic iterative optimization algorithm into a quantum version, or is it not better to directly design a new quantum algorithm? ".

Part III
Appendix

6 Aide-memoire

A qubit is a vector with two complex components

$$q = \begin{pmatrix} \alpha \\ \beta \end{pmatrix} \tag{6.1}$$

so that

$$|\alpha|^2 + |\beta|^2 = 1 \tag{6.2}$$

When we measure it we find 0 with the probability $|\alpha|^2$ or 1 with the probability $|\beta|^2$.

To simplify calculations the qubit $\begin{pmatrix} 1 \\ 0 \end{pmatrix}$ is noted $|0>$ and the qubit $\begin{pmatrix} 0 \\ 1 \end{pmatrix}$ is noted $|1>$. This is the so called *bra* notation.

A notation like $|010>$ means we consider the tensor product $|0> \otimes |1> \otimes |0>$ and is a *state*. This is then a column vector of size 2^3. A notation like $|\psi>$ is a concise way to say that ψ is a complex column vector (of any dimension) as the *ket* notation $<\varphi|$ is a complex row vector, a linear form, actually.

Inner product

Applying a ket to a bra is the *inner product* $<\varphi||\psi>$ noted $<\varphi\psi>$ and the result is a complex number. In particular we assume $<\psi\psi>= 1$, which is a generalisation of the formula 6.2.

Outer product

Applying a bra to a ket is the *outer product* $|\psi><\varphi|$ noted $|\psi\varphi|$ and the result is a matrix (product of a column vector by a row vector).

Qubits are manipulated thanks to unitary matrices, usually called operators or gates.

H gate

The name comes from Hadamard. This gate places the qubit in the "intermediate" state, which would give 0 or 1 with probabilities equal to $1/2$, if measured.

Such a gate therefore induces the transformation

$$|0> \rightarrow \frac{|0>+|1>}{\sqrt{2}} = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

In matrix representation, we have

$$H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \tag{6.3}$$

It also induces the transformation

$$|1> \rightarrow \frac{|0>-|1>}{\sqrt{2}} = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ -1 \end{pmatrix}$$

It is often used as the analogue of the classic uniform random initialization in population-based algorithms, when all qubits at the very beginning are in state $|0>$.

X gate

X for *exchange*. Sometimes called Pauli-X or NOT gate or *bit-flip*.

When a qubit 'passes' through such a gate, it becomes, as it were, its opposite: $|0>$ becomes $|1>$and $|1>$ becomes $|0>$. More generally α and β components are exchanged.

Its matrix is

$$X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \tag{6.4}$$

and to apply this gate is to perform the product

$$X \times \begin{pmatrix} \alpha \\ \beta \end{pmatrix} = \begin{pmatrix} \beta \\ \alpha \end{pmatrix}$$

This is the analogy of Not in classical computing.

CX gate

CX for *controlled exchange*.

It is also called CNOT because it is indeed a controlled 'negation'. This gate concerns two qubits. When the first one is $|1>$ an X gate is applied to the second one. But if it is $|0>$ the X gate on the second one is ignored.

So the matrix is now 4×4 :

$$CX = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{6.5}$$

From an algebraic point of view, applying this gate consists of forming a four-element ψ vector by concatenating 'vertically' those of the two qubits, then performing the product

$$CX \times \psi = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{pmatrix} \alpha_1 \\ \beta_1 \\ \alpha_2 \\ \beta_2 \end{pmatrix} = \begin{pmatrix} \alpha_1 \\ \beta_1 \\ \beta_2 \\ \alpha_2 \end{pmatrix}$$

In a circuit, this gate is often represented as

Ry gate

This operator needs a parameter θ called rotation angle. Its matrix is

$$R_y(\theta) = \begin{bmatrix} \cos\left(\frac{\theta}{2}\right) & -\sin\left(\frac{\theta}{2}\right) \\ \sin\left(\frac{\theta}{2}\right) & \cos\left(\frac{\theta}{2}\right) \end{bmatrix} \tag{6.6}$$

In the so-called Poincaré-Bloch sphere visualization it rotates around the y axis the vector representing the qubit to which it is applied, hence the name. Let's apply it to $|0>$:

$$R_y(\theta) \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} \cos\left(\frac{\theta}{2}\right) \\ \sin\left(\frac{\theta}{2}\right) \end{pmatrix}$$

So if we want to initialize a qubit in the $\begin{pmatrix} \alpha \\ \beta \end{pmatrix}$ state we just have to reset it to $|0>$ and then apply $R_y(2\arccos(\alpha))$. The β value is set to $\sin(\arccos(\alpha))$.

Pauli Z gate

This is a single-qubit rotation through π radians around the z-axis. Sometimes also noted σ_z. Its matrix is

$$Z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$$

CZ gate

The controlled Z gate, applied to two qubits. Although it is called "controled" one can not really say that a qubit "controls" another one so its graphical representation in a circuit is symmetrical:

Its matrix is

$$CZ = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix} \tag{6.7}$$

In fact, of the four possible input states, namely $|00>$, $|10>$, $|01>$ and $|11>$, only the last one is modified, transformed into $-|11>$.

Pure state, mixed state

A pure state can be represented by a vector. Qubits are pure states. However, most of the time, after some manipulations the current state is a linear combination of pure states, and is defined by a matrix, as we have seen for the H gate. However, in practice, it is easier to explicitely give the linear combination, something like

$$S = \alpha_1|01...10> + \alpha_2|10...00> + ...\alpha_n|001...0> \tag{6.8}$$

in which each "bit string" has the same length and with $\sum_{i=1}^{n} |\alpha_i|^2 = 1$ for each $|\alpha_i|^2$ is a probability. It means that if we measure the state we will find one (and just one) of the "bit string" sequences and with the associated probability. On a simulation we can effectively display the state, but not on a real quantum device.

Any quantum algorithm has to change the probabilities without being able to see them, in order to favour the desirable states before measure.

Density operator

Let us consider an ensemble of pure states ψ_j prepared with probabilities p_j (of sum 1). Then the density operator is defined by

$$\rho = \sum_j p_j \, |\psi_j\rangle \, \langle\psi_j|$$

Note this is a matrix.

Fidelity

If you consider states as probabilistic distributions, a classical estimation of the "closeness" of two states φ and ψ is the *fidelity* based on the density operator. If the states are pure the formula is simply

$$F(\varphi,\psi) = \left(tr\left(\sqrt{\sqrt{\rho}\sigma\sqrt{\rho}}\right)\right)^2) = \left(tr\left(|\sqrt{\rho\sigma}|\right)\right)^2$$

where
tr is the trace of a matrix
ρ is the density operator of φ

σ is the density operator of ψ

Although not obvious the fidelity is symmetrical. However it is not really a distance (no triangle inequality).

For pure states the formula is simply

$$F(\varphi, \psi) = |<\varphi|\psi>|^2$$

Let's consider the particular case of two qubits $q_1 = \begin{pmatrix} \alpha_1 \\ \beta_1 \end{pmatrix}$ and $q_2 = \begin{pmatrix} \alpha_2 \\ \beta_2 \end{pmatrix}$.

The fidelity is simply

$$F(q_1, q_2) = |\alpha_1^* \alpha_2 + \beta_1^* \beta_2|^2 \tag{6.9}$$

where * means "conjugate". Its value in $[0, 1]$ tell us how "similar" are these two qubits. For example if a qubit is set to $|0>$ and the other initialized thanks to a H gate, the fidelity is 0.5. But if they are both initialized by H the fidelity is 1.

7 Binarization

Let's consider a real function f defined on $S = [x_{min}, x_{max}]$ whose minimum is on x^*. If it is an artificial problem of a benchmark for optimization it is alway possible to modify it so that $x_{min} = 0$ and x^* is an integer. So if $x_{max} \leq 2^n$ the number of bits to represent in base 2 all integers of the search space S is $n - 1$.

Now each integer x of S can written

$$x = \sum_{i=0}^{n-1} b_i 2^i \tag{7.1}$$

For example the continuous function

$$f(x) \quad = \quad (1 - x)^2 \quad on \quad [0, 7]$$

becomes (we keep the same name for simplicity)

$$f(b_0, b_1, b_2) \quad = \quad (1 - b_0 - 2b_1 - 4b_2)^2 \quad on \quad \{0, 1\}^3$$

and its minimum is 0 on $(0, 0, 1)$.

Of course this transformation can be extended to more than one dimension. If we have $f(x, y)$ and if x needs n_x bits and y needs n_y bits then the resulting binary function will be

$$f\left(b_0, \ldots, b_{n_x - 1}, b_{n_x}, \ldots, b_{n_x + n_y - 1}\right) \tag{7.2}$$

8 Source codes

I had to learn Python and Qiskit (0.44.1 version) to write these codes. I didn't become an expert, but someone skilled could surely make the codes shorter and better.

I did tests on a Linux laptop using Anaconda/Jupyter, and the operating system was Ubuntu 22.04.

8.1 Grover adaptive search

```
import numpy as np
import math
import time
from qiskit_algorithms.minimum_eigensolvers \
  import NumPyMinimumEigensolver
from qiskit_optimization.algorithms \
 import GroverOptimizer, MinimumEigenOptimizer
# Works well only for Quadratic unconstrained binary optimization
from qiskit_optimization.translators \
  import from_docplex_mp
from docplex.mp.model import Model
model = Model()
# G6
x0 = model.binary_var(name="x0")
x1 = model.binary_var(name="x1")
x2 = model.binary_var(name="x2")
x3 = model.binary_var(name="x3")
x4 = model.binary_var(name="x4")
x5 = model.binary_var(name="x5")
n=6
def c(xa,xb):
  cv=4-xa -2*xb +xa*xb
  return cv

def fit(x0,x1,x2,x3,x4,x5):
  f=-(15-c(x0,x1)-c(x1,x2)-c(x2,x3)-c(x3,x4)-c(x4,x5))
  return f
#---------------- Exhaustive search, to check
start=time.process_time()
t=0
v=np.array(range(2**n))
fmax=-1000
fmin=1000
for x0_ in range(2):
  for x1_ in range(2):
  for x2_ in range(2):
```

```
    for x3_ in range(2):
    for x4_ in range(2):
      for x5_ in range(2):
      f_=fit(x0_,x1_,x2_,x3_,x4_,x5_)
      v[t]=f_
      t=t+1
      if f_<fmin:
        fmin=f_
        solutionMin=[x0_,x1_,x2_,x3_,x4_,x5_]
      if f_>fmax:
        fmax=f_
        solutionMax=[x0_,x1_,x2_,x3_,x4_,x5_]

print("Minimum ",fmin," on ", solutionMin,sep="")
print("Maximum ",fmax," on ", solutionMax,sep="")
vu=np.unique(v)
print("Possible values")
print(vu)
print("There are ",len(vu)," possible values",sep = "")
end=time.process_time()
print("Exhaustive search time = ",end-start,sep="")
#---------------------- Grover
start=time.process_time()
model.minimize(fit(x0,x1,x2,x3,x4,x5))
qp = from_docplex_mp(model)
print(qp.prettyprint())
# m will allow us to store integer values
# on the interval [-2^(m-1),2^(m-1)-1] that includes [fmin,fmax]
w1=fmin # lower bound of the fitness function
w2=fmax # upper bound
m1=math.ceil(math.log(abs(w1),2)+1)
m2=math.ceil(math.log(abs(w2)+1,2)+1)
m=max(m1,m2)
nbqubits=n+m
s= 2 # number of solutions
nbIter=math.floor((math.pi/4)*(1/math.sqrt(s/2**nbqubits) ))
print("n = ", n, ", m = ", m,", nbqubits = ", nbqubits,sep = "")
print("nbIter = ",nbIter)
# ------------------------Grover's adaptive search
if nbqubits<10: # Can run locally on a small laptop
  from qiskit.primitives import Sampler
  grover_optimizer = \
  GroverOptimizer(nbqubits, num_iterations=nbIter, \
  sampler=Sampler())

else: # IBM quantum cloud.
```

```
    # The needed API token has been saved once on the local computer
    from qiskit_ibm_runtime import Sampler
    from qiskit_ibm_runtime import QiskitRuntimeService
    service = QiskitRuntimeService(channel="ibm_quantum")
    backend = service.backend("ibmq_qasm_simulator")

    grover_optimizer =
    GroverOptimizer(nbqubits, num_iterations=nbIter, \
    sampler=Sampler(backend=backend))
results = grover_optimizer.solve(qp)
end=time.process_time()
print("Exhaustive search time = ",end-start,sep="")
print(results.prettyprint())
# ------------------------ Post-processing
res=results.samples
lres=len(res)
x=list(range(lres))
pr=list(range(lres))
for i in range(lres):
  pr[i]=res[i].probability
  s = [str(element) for element in np.int_(res[i].x)]
  x[i] =''.join(s)
import matplotlib.pyplot as plt
fig = plt.figure()
ax = fig.add_axes([0,0,1,1])
ax.bar(x,pr)
ax.set_ylabel('Probability')
ax.set_xlabel('Position')
plt.xticks(fontsize=7,rotation = 90)
plt.show()
```

8.2 Common codes

8.2.1 Functions

```
#!/usr/bin/env python
# coding: utf-8

# In[ ]:
import math
Pi = math.pi

def quadCos(x):
    cos=1-x*x/2
    return cos
```

```
def fCodeDmin(fCode):
    minimum = 0  # Default value. If you know it
    # If not you may use iterMax=Inf
    # but there is a risk of infinite loop

    match fCode:
        case -1:
            D = 1
        case 0:
            D = 2
        case 1:
            D = 2
        case 2:
            D = 2
        case 3:  # Unimodal
            D = 2
        case 4:
            D = 3
        case 5:
            D = 4
            minimum = -5
        case 51:
            D = 6
            minimum = -5
        case 6:  # Unimodal
            D = 5
            minimum = 0
        case 7:  # Unimodal
            D = 6
        case 8: # sin(u)*(1-u)
            D = 3
            minimum = -3.94
        case 9:  # 2D Rastrigin on [0,7]^2
            D = 6
        case 91: # 2D Rastrigin as quadratic function
            D = 6

        case 10:  # 2D Rastrigin on[0,15]^2
            D = 8

        # A set of simple functions, just to test
        # a possible bias
        case 11: # Minimum 0 on 010
            D=3
        case 12: # Minimum on 100
            D=3
```

```
        case 13: # Minimum on 111
            D=3
        case 14: # Minimum on 000
            D=3
        case 15:
            D=6
    return D,minimum


def fit(x, fCode):  # Fitness function
    b0 = x[0]

    match fCode:
        case -1: # Just to test
            f=b0
        case 0:  # Solution 10
            b1 = x[1]
            f = (b0-1)**2 + b1
        case 1:
            b1 = x[1]
            f = (b0+2*b1-1)**2 * (3-b0-2*b1) + (3-b0-2*b1)/4
        case 2:
            b1 = x[1]
            f = (1-2*b0-4*b1)**2 * (3-b0-2*b1)

        case 3:  # Unimodal
            b1 = x[1]
            f = (2*b0+4*b1-2)**2

        case 4:  # Solutions 110 101
            b1 = x[1]
            b2 = x[2]
            f = (1-2*b0-4*b1+b2)**2 * (3-b0-2*b1+b2)

        case 5:  # G4
            b1 = x[1]
            b2 = x[2]
            b3 = x[3]
            f = -11+(4-b0-2*b1+b0*b1) + (4-b1-2*b2+b1*b2)+(4-b2-2*b3+b2*b3)

        case 51:  # G6. Solutions 111111 011111. Difficult
            b1 = x[1]
            b2 = x[2]
            b3 = x[3]
            b4 = x[4]
            b5 = x[5]
```

```
        f = -15+(4-b0-2*b1+b0*b1) + (4-b1-2*b2+b1*b2)+(4-b2-2*b3+b2*b3) +\
            (4-b3-2*b4+b3*b4) + (4-b4-2*b5+b4*b5)

    case 6:  # Unimodal. Solution 10000
        b1 = x[1]
        b2 = x[2]
        b3 = x[3]
        b4 = x[4]
        f = (1-b0-2*b1-4*b2-8*b3-16*b4)**2

    case 7:  # Unimodal. Solution 100000
        b1 = x[1]
        b2 = x[2]
        b3 = x[3]
        b4 = x[4]
        b5 = x[5]
        f = (1-b0-2*b1-4*b2-8*b3-16*b4-32*b5)**2

    case 8:  # Solution 111
        b1 = x[1]
        b2 = x[2]
        u = b0+2*b1+4*b2
        f = math.sin(u)*(1-u)

    case 9:  # 2D Rastrigin on [0,7]^2
        # Minimum 0 on [3,5]
        # Solution 110 101
        b1 = x[1]
        b2 = x[2]
        # second coordinate
        b3 = x[3]
        b4 = x[4]
        b5 = x[5]

        u1 = b0+2*b1+4*b2 - 3
        u2 = b3+2*b4+4*b5 - 5
        f = 20 + u1*u1-10*math.cos(2*Pi*u1) \
            + u2*u2-10*math.cos(2*Pi*u2)

    case 91: # 2D Rastrigin as quadratic function
        b1 = x[1]
        b2 = x[2]
        b3 = x[3]
        b4 = x[4]
        b5 = x[5]
        u1 = b0+2*b1+4*b2 - 3
```

```
            u2 = b3+2*b4+4*b5 - 5
            c1=quadCos(2*Pi*u1)
            c2=quadCos(2*Pi*u2)
            f =20 + u1*u1-10*c1  + u2*u2-10*c2

        case 10:  # 2D Rastrigin on [0,15]^2
            # Centered on [3,5]
            b1 = x[1]
            b2 = x[2]
            b3 = x[3]
            # second coordinate
            b4 = x[4]
            b5 = x[5]
            b6 = x[6]
            b7 = x[7]
            u1 = b0+2*b1+4*b2+8*b3 - 3
            u2 = b4+2*b5+4*b6+8*b7 - 5
            f = 20 + u1*u1-10*math.cos(2*Pi*u1) \
                + u2*u2-10*math.cos(2*Pi*u2)

        # Just to detect possible biases
        case 11: # Solution 010
            f=b0+1-x[1]+x[2]
        case 12: # Solution 100
            f=1-b0+x[1]+x[2]
        case 13: # Solution 111
            f=3-b0-x[1]-x[2]
        case 14: # Solution 000
            f=b0+x[1]+x[2]

    return f
```

8.2.2 Tools

```
#!/usr/bin/env python
# coding: utf-8

# In[ ]:

import numpy as np
import math
Pi = math.pi
Inf = math.inf
from statistics import mean
```

```
from qiskit import *

from random import random, uniform, sample, randrange, randint, choice
from qiskit.quantum_info import DensityMatrix

from qiskit import Aer
from qiskit.extensions import Initialize

def IBM_backends_list():
    # List of available backends
    from qiskit_ibm_runtime import QiskitRuntimeService
    backends = QiskitRuntimeService().backends()
    for i in range(len(backends)):
        print(backends[i])

#-----------------------------------------------------------
def initPos(rank1,rank2,qc):  # Init explorer swarm
    qc.h(range(rank1,rank2))

def initPos2(rank1,rank2,qc):
    for i in range(rank1,rank2):
        theta=uniform(0,2*Pi)
        qc.ry(theta,i)

def mostProbable(counts,tolerance):
    values = list(counts.values())
    keys = list(counts.keys())
    # With "local" run, the length of the keys
    # is exactly the number of measured qubits
    proba=np.array(values)/sum(values)
    probaMax=max(proba)
    # Find more or less equivalent ones
    sList=[]
    for i in range(len(values)):
        if values[i]>= tolerance*probaMax:
            sList.append(keys[i])
    # Random choice
    string=choice(sList)
    return  string,probaMax

def mostProbableIBM(qd0,tolerance, measured):
    # Extract from a quasi-distribution
    # (coming from the IBM cloud, for example)
    proba=list(qd0.values()) # Probabilities
    keys=list(qd0.keys()) # Bit string (coded as integer)
    '''
```

```
    On the IBM cloud the length of the key
    as bit string may need leading zeros
    so that the string has the right length
    (i.e.) the number of measured qubits.
    '''

    probaMax=max(proba)

    # Find more or less equivalent ones
    sList=[]
    for i in range(len(proba)):
        if proba[i] >= tolerance*probaMax:
            sList.append(keys[i])

    key=choice(sList) # Random choice
    string = format(key,'b') # integer => binary string
    ls=len(string)
    if ls<measured:
        string=string.zfill(measured)

    return string,probaMax

def qubitDensityIBM(qd0,measured,i):
    keys=list(qd0.keys())
    values=list(qd0.values())

    # For qubit i, probability to be 0
    proba0=0
    for k in range(len(keys)):
        string = format(keys[k],'b')
        ls=len(string)
        if ls<measured: # Complete with leading zeros
            string=string.zfill(measured)
        ind=len(string)-i-1 # Because litte-endian notation
        if string[ind]=='0':
                proba0=proba0+values[k]

        qDensity=DensityMatrix([[proba0,0],[0,1-proba0]])
    return qDensity

def qubitDensity(counts,i):
    keys=list(counts.keys())
    values=np.array(list(counts.values()))
    values=values/sum(values)

    # For qubit i, probability to be 0
```

```
    proba0=0
    for k in range(len(keys)):
        string = keys[k]
        ind=len(string)-i-1 # Because litte-endian notation
        if string[ind]=='0':
                proba0=proba0+values[k]

        qDensity=DensityMatrix([[proba0,0],[0,1-proba0]])
    return qDensity

def diagSquare(M):
    diag=np.sqrt(np.diagonal(M))
    return diag

def assignProba0(qc,i,dm,p):
    # dm = density matrix of a qubit
    # p = desired probability for |0>
    #  Just a bit of algebra
    p0=np.array(dm)[0][0] # Initial probability
    A=math.sqrt(p0*p)
    B=math.sqrt(max(p0*p -p-p0+1,0)) # It may be slightly <0
    # Because numerical instability
    x1= max(min(A+B,1),-1)
    x2= max(min(A-B,1),-1)
    theta1=2*math.acos(x1)
    theta2=2*math.acos(x2)
    theta= min(theta1,theta2)
    # Then you have to apply Ry(theta,i)
    qc.ry(theta,i)
    return qc

def numberDraws(pr,m):
    '''
    Number of draws with replacement.
    m = number of objects
    pr = desired probability to draw each one
    '''
    pt=0
    n=100
    while pt<pr:
        n=n+100 # Should be +1, but faster
        pt=probaEach(m,n)
        # To cope with the numerical instability
        # Warning: may induce infinite loop
        # or incorrect results if m is too big
        if pt<0 or pt>1:
```

```
            pt=0
        return n

    def probaEach(m,nd):
        '''
        Probability to draw each object at least once.
        m = number of objects
        nd = number of draws
        '''
        p=0
        for i in range(m+1):
            comb=combin(m,i)
            #print("comb",comb)
            p=p+((-1)**i) * comb * ((i/m)**nd)
            #print("p",p)
        return p

    def combin(m, k):
        "Combine m objects k by k"
        " m <= 410 "
        if k > m // 2:
            k = m - k
        result = 1
        for i in range(1, k + 1):
            result *= m
            result //= i
            m -= 1
        return result

    def dispCircuit(qc):
        qcDec = qc.decompose().decompose().decompose().decompose()
        # print(qcDec)
        qcDec.draw()

    def init01(qc,bitstring):
        D=len(bitstring)
        for d in range(D):
            bit=bitstring[D-1-d]
            if bit=="1":
                vector=[0,1]
            else:
                vector=[1,0]
            qc.initialize(vector,d)


#!/usr/bin/env python
```

```
# coding: utf-8

# In[ ]:
import numpy as np
import math
Pi = math.pi

from random import random, uniform, sample, randrange, randint, choice
from qiskit.quantum_info import state_fidelity

def initVel(N, D, vMax):  # (N,D,vMax): # 'Velocity' initialisation
    v = []
    for _ in range(N*D):
        v.append(uniform(-vMax, vMax))
    return v

def updateVelNoSv(N,D,nqbits,v,counts,w,c2,vMax):  # Velocity update
    S = []
    for i in range(N):
        for d in range(D):
            i1 = D*i+d  # current qubit of the position
            i2 = nqbits-D + d  # current qubit of g
            ifid2 = infidNoSv(i1, i2, counts)
            v[i1] = w*v[i1] + uniform(0, c2)*ifid2
            if abs(v[i1]) > vMax:
                print("   Warning, v", v[i1])

            S.append(mapping(v[i1]))
    v = np.array(v)
    S = np.array(S)
    return v, S

def updateVelNoSvIBM(N,D,nqbits,v, qd,w,c2,vMax,measured):
# Velocity update
    S = []
    for i in range(N):
        for d in range(D):
            i1 = D*i+d  # current qubit of the position
            i2 = nqbits-D + d  # current qubit of g
            ifid2=infidNoSvIBM(i1,i2,qd,measured)
            v[i1] = w*v[i1] + uniform(0, c2)*ifid2
            if abs(v[i1]) > vMax:
                print("   Warning, v", v[i1])

            S.append(mapping(v[i1]))
    v = np.array(v)
```

```
    S = np.array(S)
    return v, S

def copyPos(i1, i2,qc,D):
    rank1 = D*i1
    rank2 = D*i2
    for d in range(D):
        qc.reset(rank2+d)
        qc.cx(rank1+d, rank2+d)

def updatePos(S,N,D,qc):  # 'Position' update
    for i in range(N):
        for d in range(D):
            j = D*i+d
            qc.ry(Pi*S[j],j)

def dispCircuit(qc):
    qcDec = qc.decompose().decompose().decompose().decompose()
    #qcDec.draw()
    print(qcDec)

def infidNoSv(i1,i2,counts):
    sv1=qubitStateCounts(counts,i1)
    sv2=qubitStateCounts(counts,i2)
    ifid = 1 - state_fidelity(sv1, sv2)  # in [0,1]
    ifid = 2*ifid-1  # in [-1,1]
    ifid = round(ifid)  # in {-1,0,1}
    return ifid

def infidNoSvIBM(i1,i2,qd,measured):
    sv1=qubitStateCountsIBM(qd,i1,measured)
    sv2=qubitStateCountsIBM(qd,i2,measured)
    ifid = 1 - state_fidelity(sv1, sv2)  # in [0,1]
    ifid = 2*ifid-1  # in [-1,1]
    #ifid = round(ifid)  # in {-1,0,1}
    return ifid

def qubitStateCountsIBM(qd0,i,measured):
    keys=list(qd0.keys())
    values=list(qd0.values())

    # For qubit i, probability to be 0
    proba0=0
    for k in range(len(keys)):
        string = format(keys[k],'b')
        ls=len(string)
```

```
        if ls<measured: # Complete with leading zeros
            string=string.zfill(measured)
        ind=len(string)-i-1 # Because litte-endian notation
        if string[ind]=='0':
                proba0=proba0+values[k]

        p=[math.sqrt(proba0),math.sqrt(1-proba0)]
    return p

def qubitStateCounts(counts,i):
    keys=list(counts.keys())
    values=np.array(list(counts.values()))
    values=values/sum(values)

    # For qubit i, probability to be 0
    proba0=0
    for k in range(len(keys)):
        string = keys[k]
        ind=len(string)-i-1 # Because litte-endian notation
        #ind=i
        if string[ind]=='0':
                proba0=proba0+values[k]

        p=[math.sqrt(proba0),math.sqrt(1-proba0)]
    return p

def mapping(u):
        a=2
    sig = 2 / (1 + math.exp(-a*u)) -1
    return sig
```

8.3 Global best

```
#!/usr/bin/env python
# coding: utf-8

# In[ ]:


'''
Hybrid Quantum PSO for binary problems
by Maurice.Clerc@>WriteMe.com
Last update: 2024-01-13
This version does not use the state vector,
so it is more suitable for a real quantum device
```

```
Just for a proof of concept:
    - global best topology
    - mimicking the classical binary PSO

May need (Linux)
echo 1 | sudo tee /proc/sys/vm/overcommit_memory

'''
from Tools2 import *
from Tools1 import *
from Functions import *
np.set_printoptions(precision=3)
from qiskit.quantum_info import state_fidelity

#-------------------------------------------------------------
max_qubits = 17  # The max for my computer
IBM = False  # Note: it will be forced to True
             # if more qubits than max_qubits

fCode = 8  # *** Here choose the function

runMax = 100
tMax = 4 # Number of iterations for each run
#  You may set it to Inf but
#  WARNING: possible infinite loop
verbose=False
plotCircuit=False
restore=False # Just for test
#-------------------------------------
# Parameters
N = 2 # swarm size
w = 0.721  #  Inertia weight
c2 = 1.193 #  Cognitive/social coefficient
vMax=c2/(1-w)
tolerance = 0.95  # To define "equivalent" probabilities

#-------------------------------------
D, minimum = fCodeDmin(fCode)
nqbits = N*D +D
if nqbits <= 8:
    nshots = numberDraws(0.9, 2**nqbits)
else:  # User-defined
    nshots =2048 # 2**12
print(nshots, "nshots")
```

```
nbits = nqbits  # To save the fitness value
print("nqbits",nqbits)
allqbits = list(range(nqbits))
measured = N*D #nqbits
measuredQbits=list(range(N*D))

if verbose:
    print("Function", fCode)
    print("Dimension", D)
    print(nqbits, "qubits")
IBM = nqbits > max_qubits or IBM
if IBM:
    print("IBM cloud")
    from qiskit_ibm_runtime import Sampler, Session, QiskitRuntimeService
        from qiskit_ibm_runtime import Estimator
    service = QiskitRuntimeService(channel="ibm_quantum")
    backend = service.backend("ibmq_qasm_simulator")
else:
    backend = Aer.get_backend('statevector_simulator')
# ------------------------------------------------------------ Runs
fMinBest = Inf  # Final best value over all runs
FEtot = 0  # Total number of evaluations
success = 0
FEsuccess = []

for run in range(runMax):
    print("------------------------- Run ", run+1)
    FEs = 0  # Number of evaluations, for information
    fMin = Inf
    iBest = N  # Gobal best
    t = 0
    #plusMinus = [1, -1]
    fList = []  # Just for information
    stop=False

    while not stop:
        t=t+1
        if verbose:
            print("-------------------------", "Iteration", t)
        q = QuantumRegister(nqbits)
        c = ClassicalRegister(measured)
        qc = QuantumCircuit(q, c)
        if t == 1:
            initPos(0, N*D, qc)
            v = initVel(N, D, vMax)  # Random velocity
        else:
```

```
        if restore:
            init01(qc, bitstring)
        if IBM:
            v,S=updateVelNoSvIBM(N,D,nqbits,v, qd,w,c2,vMax,measured)
        else:
            v,S=updateVelNoSv(N,D,nqbits,v,counts,w,c2,vMax)

        updatePos(S, N, D, qc)

    qc.measure(measuredQbits, measuredQbits)
    if plotCircuit:
        qc.barrier(range(nqbits))  # Just to better see the circuit
        dispCircuit(qc)

    # ----------
    if IBM:
        with Session(backend=backend):
            sampler = Sampler()
            result = sampler.run(qc, shots=nshots).result()
            qd = result.quasi_dists[0]
            bitstring, proba = mostProbableIBM(qd, tolerance, measured)
    else:
        job = execute(qc, backend, shots=nshots)
        result = job.result()
        counts = result.get_counts(qc)
        bitstring,proba = mostProbable(counts,tolerance)

    if verbose:
        print("Most probable", bitstring, "proba:", proba)

    # For each q-particle
    #  evaluate its fitness and compare
    for n in range(N):  # range(N) ??  N-1 ??
        x = np.zeros(D, dtype=int)
        ls = len(bitstring)
        for d in range(D):
            # (little-endian notation)
            x[d] = int(bitstring[ls-d-1 - n*D])
        f = fit(x, fCode)
        FEs = FEs+1
        fList.append(f)  # For information
        #print("f",f)
        # If improvement
        if f < fMin:
            xBest = x  # Global best position
            if verbose:
```

```
                    print("Improvement. xBest=", xBest, fMin, "=>", f)
                fMin = f
                copyPos(n, iBest, qc, D)  # To global best q-particle g
                if fMin < fMinBest:
                    fMinBest = fMin
                    xBestBest = xBest

        stop = t >= tMax or fMin <= minimum
        if t >= tMax:
            print("STOP t>=", tMax)
        if fMin <= minimum:
            print("STOP fBest<=", minimum)
            success=success+1
            FEsuccess.append(FEs)
        if plotCircuit:
            dispCircuit(qc)

        if not stop:
            FEtot = FEtot+FEs
            if verbose:
                print("After ", t, "iterations:")
                print("  and", FEs, "evaluations")
                print(xBest, " =>", fMin)

print("================================ Final result ")
print("Function", fCode)
print("Dimension", D)
print("Swarm size",N)
print("Number of qubits",nqbits)
print("Runs", runMax)
print("Iterations/run ", tMax)
print(nshots, "shots")
print("restore",restore)
print("Total number of evaluations", FEtot)
print("w,c",w,c2)
print("Best result:")
print(xBestBest, " =>", fMinBest)
print("Success rate", success/runMax)
print("Success rate with random search", 1/2**D)

if success>0:
        mFE=round(mean(FEsuccess))
    print("Mean number of evaluations for successful runs:",mFE )


# In[ ]:
```

8.4 Local best

```
#!/usr/bin/env python
# coding: utf-8

# In[2]:


'''
Hybrid Quantum PSO for binary problems
by Maurice.Clerc@>WriteMe.com
Last update: 2024-01-15

Just for a proof of concept:
    - local best topology
    - mimicking the classical binary PSO

May need (Linux)
echo 1 | sudo tee /proc/sys/vm/overcommit_memory

'''
from qiskit.quantum_info import state_fidelity
from Functions import *
from Tools1 import *
from Tools2 import *


def neighbour(N, K, n):  # Neighbours of n. Random topology
    OK = False
    listn = list(range(0, N))
    # print("listn",listn)
    listn.remove(n)  # Not n itself
    #print("N,K,n,listn , n",N,K,n,listn)
    neigh = []
    for _ in range(K):
        if listn:  # As long as not empty
            ni = choice(listn)  # random choice
            neigh.append(ni)
            listn.remove(ni)  # To avoid duplicates
            # print('ni',ni)
    return neigh


def updateVel(N, K, D, nqbits, v, counts, w, c1, c2, vMax, fPrev):
# Velocity update
    S = []
```

```
    for i in range(N):
        # Find the local best
        neigh = neighbour(N, K, i)
        fPrev_n = [fPrev[i] for i in neigh]
        p_rank = neigh[np.array(fPrev_n).argmax()]

        for d in range(D):
            i1 = D*i+d  # current qubit of the position
            i2 = p_rank*D+d  # local best
            i3 = N*D+i*D+d   # previous best
            ifid1 = infidNoSv(i1, i3, counts)
            ifid2 = infidNoSv(i1, i2, counts)
            v[i1] = w*v[i1] + uniform(0, c2)*ifid2 \
                + uniform(0, c1)*ifid1
            if abs(v[i1]) > vMax:
                print("   Warning, v", v[i1])

            S.append(mapping(v[i1]))
    v = np.array(v)
    S = np.array(S)
    return v, S


def updateVelIBM(N, K, D, nqbits, v, qd, w, c1, c2, vMax, fPrev, measure):
# Velocity update
    S = []
    for i in range(N):
        # Find the local best
        neigh = neighbour(N, K, i)
        fPrev_n = [fPrev[i] for i in neigh]
        p_rank = neigh[np.array(fPrev_n).argmax()]

        for d in range(D):
            i1 = D*i+d  # current qubit of the position
            i2 = p_rank*D+d  # local best
            i3 = N*D+i*D+d   # previous best
            ifid1 = infidNoSvIBM(i1, i3, qd, measured)
            ifid2 = infidNoSvIBM(i1, i2, qd, measured)
            v[i1] = w*v[i1] + uniform(0, c2)*ifid2 \
                + uniform(0, c1)*ifid1
            if abs(v[i1]) > vMax:
                print("   Warning, v", v[i1])

            S.append(mapping(v[i1]))
    v = np.array(v)
    S = np.array(S)
```

```
    return v, S


# ------------------------------------------------------------------
np.set_printoptions(precision=3)
# ------------------------------------------------------------------
max_qubits = 17  # The max for my computer
IBM = False  # Note: it will be forced to True
# if more qubits than max_qubits

fCode = 9  # *** Here choose the function

runMax = 100
tMax = 10  # Number of iterations for each run
#  You may set it to Inf but
#  WARNING: possible infinite loop
verbose = False
plotCircuit = False
# ---------------------------------
# Parameters
N = 2  # swarm size
K = 1
w = 0.721  # Inertia weight
c1 = 1.193  # Cognitive coefficient
c2 = 1.193  # Social coefficient
vMax = (c1+c2)/(1-w)
tolerance = 0.95  # To define "equivalent" probabilities

# ---------------------------------
D, minimum = fCodeDmin(fCode)
'''
N*D qubits explorer swarm
N*D qubits memory swarm
'''
nqbits = 2*N*D
if nqbits <= 8:
    nshots = numberDraws(0.9, 2**nqbits)
else:  # User-defined
    nshots = 2048  # 2**12
print(nshots, "nshots")

nbits = nqbits  # To save the fitness value
print("nqbits", nqbits)
allqbits = list(range(nqbits))
measured = N*D  # nqbits
measuredQbits = list(range(N*D))
```

```
if verbose:
    print("Function", fCode)
    print("Dimension", D)
    print(nqbits, "qubits")
IBM = nqbits > max_qubits or IBM
if IBM:
    print("IBM cloud")
    from qiskit_ibm_runtime import Sampler, Session, QiskitRuntimeService, Estim
        from qiskit_ibm_runtime import Estimator
    service = QiskitRuntimeService(channel="ibm_quantum")
    backend = service.backend("ibmq_qasm_simulator")
else:
    backend = Aer.get_backend('statevector_simulator')
# ------------------------------------------------------------------ Runs
fMinBest = Inf  # Final best value over all runs
FEtot = 0  # Total number of evaluations
success = 0
FEsuccess = []

for run in range(runMax):
    print("------------------------- Run ", run+1)
    FEs = 0  # Number of evaluations, for information
    fMin = Inf
    # iBest = N  # Gobal best
    t = 0
    fPrev = Inf*np.ones(N)
    #plusMinus = [1, -1]
    fList = []  # Just for information
    stop = False

    while not stop:
        t = t+1
        if verbose:
            print("------------------------------", "Iteration", t)
        q = QuantumRegister(nqbits)
        c = ClassicalRegister(measured)
        qc = QuantumCircuit(q, c)
        if t == 1:
            initPos(0, N*D, qc)
            v = initVel(N, D, vMax)  # Random velocity
        else:
            # Restore last most probable state
            init01(qc,bitstring)
            # Update
            if IBM:
```

```
            v, S = updateVelIBM(N, K, D, nqbits, v, qd,
                                w, c1, c2, vMax, fPrev, measured)
        else:
            v, S = updateVel(N, K, D, nqbits, v, counts,
                             w, c1, c2, vMax, fPrev)

        updatePos(S, N, D, qc)

    qc.measure(measuredQbits, measuredQbits)
    if plotCircuit:
        qc.barrier(range(nqbits))  # Just to better see the circuit
        dispCircuit(qc)

    # ----------
    if IBM:
        with Session(backend=backend):
            sampler = Sampler()
            result = sampler.run(qc, shots=nshots).result()
            qd = result.quasi_dists[0]
            bitstring, proba = mostProbableIBM(qd, tolerance, measured)
    else:
        job = execute(qc, backend, shots=nshots)
        result = job.result()
        counts = result.get_counts(qc)
        bitstring, proba = mostProbable(counts, tolerance)

    if verbose:
        print("Most probable", bitstring, "proba:", proba)

    # For each q-particle
    #  evaluate its fitness and compare
    for n in range(N):
        x = np.zeros(D, dtype=int)
        #ls = len(bitstring)
        for d in range(D):
            # (little-endian notation)
            x[d] = int(bitstring[D-d-1 - n*D])
        f = fit(x, fCode)
        FEs = FEs+1
        fList.append(f)  # For information

        # If local improvement update the previous best
        if f < fPrev[n]:
            fPrev[n] = f
            copyPos(n, N+n, qc, D)
```

```
            # If global improvement
            if f < fMin:
                xBest = x  # Global best position
                if verbose:
                    print("Improvement. xBest=", xBest, fMin, "=>", f)
                fMin = f
                # copyPos(n, iBest, qc, D)  # To global best q-particle g
                if fMin < fMinBest:
                    fMinBest = fMin
                    xBestBest = xBest

        stop = t >= tMax or fMin <= minimum
        if t >= tMax:
            print("STOP t>=", tMax)
        if fMin <= minimum:
            print("STOP fBest<=", minimum)
            success = success+1
            FEsuccess.append(FEs)
        if plotCircuit:
            dispCircuit(qc)

        if not stop:
            FEtot = FEtot+FEs
            if verbose:
                print("After ", t, "iterations:")
                print("  and", FEs, "evaluations")
                print(xBest, " =>", fMin)

print("================================ Final result ")
print("Function", fCode)
print("Dimension", D)
print("Swarm size", N)
print("Neighborhood size", K)
print("Number of qubits", nqbits)
print("Runs", runMax)
print("Iterations/run ", tMax)
print(nshots, "shots")
print("Total number of evaluations", FEtot)
print("w,c1,c2", w, c1, c2)
print("Best result:")
print(xBestBest, " =>", fMinBest)
print("Success rate", success/runMax)
print("Success rate with random search", 1/2**D)

if success>0:
        mFE=round(mean(FEsuccess))
```

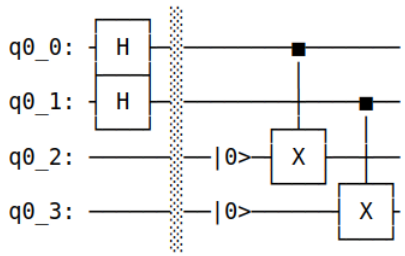


Figure 8.1: Copying qubits. $2 \Rightarrow 2$ example

```
print("Mean number of evaluations for successful runs:",mFE )
```

8.5 Copying a state

Below is a Qiskit code to temporarily copy the state of a set of qubits (here two) to another one. It is possible only if the system accepts reset. This feature can be used to apply rules like

if $f(x) \leq f(best)$ then $best = x$

```
#  2 qubits => 2 qubits example
from qiskit import *
from qiskit.quantum_info import Statevector
# Define a circuit
q = QuantumRegister(4)
qc = QuantumCircuit(q)
qc.h(0)
qc.h(1)
qc.barrier(q)  # For clarity
qc.reset([2, 3])
qc.cx(0, 2)
qc.cx(1, 3)
print(qc)
# Note:This visualization is possible
#      only in a simulation
sv = Statevector.from_instruction(qc)
sv.draw('latex')
```

By running this code we generate the circuit of the figure 8.1, and the state vector (visible only in a simulation, not on a real quantum device) is

$$\frac{1}{2}|0000> + \frac{1}{2}|0101> + \frac{1}{2}|1010> + \frac{1}{2}|1111>$$

Instead of 16 states we have only 4, because qubits 2 and 3 are always equal to qubits 0 and 1.

References

Baritompa, W. P., D. W. Bulger, and G. R. Wood (Jan. 2005). "Grover's Quantum Algorithm Applied to Global Optimization." In: *SIAM Journal on Optimization* 15.4, pp. 1170–1184. ISSN: 1052-6234. DOI: 10.1137/040605072. URL: https://epubs.siam.org/doi/abs/10.1137/040605072 (visited on 01/04/2020).

Blackwell, Tim and J. Branke (2004). "Multi-Swarm Optimization in Dynamic Environments." In: *Applications of Evolutionary Computing*. Ed. by G. R. Raidl. Vol. 3005 LNCS. Springer, pp. 488–599.

Bužek, V. and M. Hillery (Sept. 1996). "Quantum copying: Beyond the no-cloning theorem." In: *Physical Review A* 54.3. Publisher: American Physical Society, pp. 1844–1852. DOI: 10.1103/PhysRevA.54.1844. URL: https://link.aps.org/doi/10.1103/PhysRevA.54.1844 (visited on 08/29/2023).

Clerc, M. and J. Kennedy (Feb. 2002). "The particle swarm - explosion, stability, and convergence in a multidimensional complex space." In: *IEEE Transactions on Evolutionary Computation* 6.1, pp. 58–73.

Clerc, Maurice (2005). *Binary Particle Swarm Optimisers: toolbox, derivations, and mathematical insights*. Tech. rep. URL: http://hal.archives-ouvertes.fr/hal-00122809/en/.

Fallahi, Saeed and Mohamadreza Taghadosi (Aug. 2022). "Quantum-behaved particle swarm optimization based on solitons." en. In: *Scientific Reports* 12.1. Number: 1 Publisher: Nature Publishing Group, p. 13977. ISSN: 2045-2322. DOI: 10.1038/s41598-022-18351-0. URL: https://www.nature.com/articles/s41598-022-18351-0 (visited on 08/04/2023).

Flori, Arnaud, Hamouche Oulhadj, and Patrick Siarry (June 2022). "QUAntum Particle Swarm Optimization: an auto-adaptive PSO for local and global optimization." en. In: *Computational Optimization and Applications* 82.2, pp. 525–559. ISSN: 1573-2894. DOI: 10.1007/s10589-022-00362-2. URL: https://doi.org/10.1007/s10589-022-00362-2 (visited on 05/21/2022).

Hua, Fei et al. (Nov. 2022). *Exploiting Qubit Reuse through Mid-circuit Measurement and Reset*. arXiv:2211.01925 [quant-ph]. DOI: 10.48550/arXiv.2211.01925. URL: http://arxiv.org/abs/2211.01925 (visited on 01/19/2023).

Kennedy, J. and R. C. Eberhart (1997). "A discrete binary version of the particle swarm algorithm." In: *1997 IEEE International Conference on Systems, Man, and Cybernetics, 1997. 'Computational Cybernetics and Simulation'*. 4104–4108 vol.5.

Lee, Sangwook et al. (Sept. 2008). "Modified binary particle swarm optimization." en. In: *Progress in Natural Science* 18.9, pp. 1161–1166. ISSN: 1002-0071. DOI: 10.1016/j.pnsc.2008.03.018. URL: https://www.sciencedirect.com/science/article/pii/S1002007108002189 (visited on 08/10/2023).

Liu, Guoqiang et al. (Apr. 2019). "A Quantum Particle Swarm Optimization Algorithm with Teamwork Evolutionary Strategy." en. In: *Mathematical Problems in Engineering* 2019. Publisher: Hindawi, e1805198. ISSN: 1024-123X. DOI: 10.1155/2019/1805198. URL: https://www.hindawi.com/journals/mpe/2019/1805198/ (visited on 11/14/2023).

Liu, Jing, Wenbo Xu, and Jun Sun (2005). "Quantum-behaved particle swarm optimization with mutation operator." In: *2005. ICTAI 05. 17th IEEE International Conference on Tools with Artificial Intelligence*, 4 pp.

Marinescu, Dan C. and Gabriela M. Marinescu (Jan. 2012). "Classical and Quantum Information." en. In: *Classical and Quantum Information*. Ed. by Dan C. Marinescu and Gabriela M. Marinescu. Boston: Academic Press, pp. 1–131. ISBN: 978-0-12-383874-2. DOI: 10.1016/B978-0-12-383874-2.00001-1. URL: https://www.sciencedirect.com/science/article/pii/B9780123838742000011 (visited on 08/01/2023).

Mastriani, Mario (2022). "Quantum Stretching: a quasi-copy technique of arbitrary qubits for quantum internet." en. In: URL: https://www.academia.edu/39888107/Quantum_Stretching_a_quasi_copy_technique_of_arbitrary_qubits_for_quantum_internet (visited on 11/03/2023).

Mikki, S. and A. A. Kishk (2005). "Investigation of the quantum particle swarm optimization technique for electromagnetic applications." In: *Antennas and Propagation Society International Symposium, 2005 IEEE*, 45–48 vol. 2A.

Mikki, S. M. and A. A. Kishk (Oct. 2006). "Quantum Particle Swarm Optimization for Electromagnetics." In: *IEEE Transactions on Antennas and Propagation* 54.10, pp. 2764–2775.

Morimoto, Kohei et al. (Nov. 2023). *Continuous optimization by quantum adaptive distribution search*. arXiv:2311.17353 [quant-ph]. DOI: 10.48550/arXiv.2311.17353. URL: http://arxiv.org/abs/2311.17353 (visited on 12/11/2023).

Oliveira, L. D. de et al. (2006). "Particle Swarm and Quantum Particle Swarm Optimization Applied to DS/CDMA Multiuser Detection in Flat Rayleigh Channels." In: *2006 IEEE Ninth International Symposium on Spread Spectrum Techniques and Applications*, pp. 133–137.

Ortega Ballesteros, Gerard (Jan. 2021). "Quantum algorithms for function optimization." eng. In: Accepted: 2021-11-23T12:28:16Z. URL: https://diposit.ub.edu/dspace/handle/2445/181432 (visited on 10/26/2023).

PSC (2024). *Particle Swarm Central*. URL: http://particleswarm.info (visited on 01/13/2024).

Shuyuan, Yang, Wang Min, and Jiao Licheng (June 2004). "A Quantum Particle Swarm Optimization." In: *IEEE Congress on Evolutionary Computation*. Portland, Oregon, USA: IEEE Press, pp. 320–324.

Soloviev, Vicente P., Concha Bielza, and Pedro Larrañaga (June 2021). "Quantum-Inspired Estimation Of Distribution Algorithm To Solve The Travelling Salesman Problem." In: *2021 IEEE Congress on Evolutionary Computation (CEC)*, pp. 416–425. DOI: 10.1109/CEC45853.2021.9504821. URL: https://ieeexplore.ieee.org/document/9504821 (visited on 12/11/2023).

Sun, Jun, Wei Fang, et al. (2012). "Quantum-behaved particle swarm optimization: analysis of individual particle behavior and parameter selection." eng. In: *Evolutionary Computation* 20.3, pp. 349–393. ISSN: 1530-9304. DOI: 10.1162/EVCO_a_00049.

Sun, Jun, Choi-Hong Lai, and Xiao-Jun Wu (2019). *Particle Swarm Optimisation: Classical and Quantum Perspectives*. en. URL: https://www.crcpress.

com/Particle-Swarm-Optimisation-Classical-and-Quantum-Perspectives/Sun-Lai-Wu/p/book/9780367381936 (visited on 12/14/2019).

Sun, Jun, Wenbo Xu, and Bin Feng (2004). "A global search strategy of quantum-behaved particle swarm optimization." In: *2004 IEEE Conference on Cybernetics and Intelligent Systems*. Volume 1, 1-3 Dec. 2004, 111–116 vol.1.

Wikipedia (July 2023). *W state*. en. Page Version ID: 1166599331. URL: https://en.wikipedia.org/w/index.php?title=W_state&oldid=1166599331 (visited on 08/12/2023).

Wootters, W. K. and W. H. Zurek (Oct. 1982). "A single quantum cannot be cloned." en. In: *Nature* 299.5886. Number: 5886 Publisher: Nature Publishing Group, pp. 802–803. ISSN: 1476-4687. DOI: 10.1038/299802a0. URL: https://www.nature.com/articles/299802a0 (visited on 08/29/2023).

Yang, Shuyuan, Min Wang, and Licheng jiao (2004). "A quantum particle swarm optimization." In: *Congress on Evolutionary Computation, 2004. CEC2004.* 320–324 Vol.1.