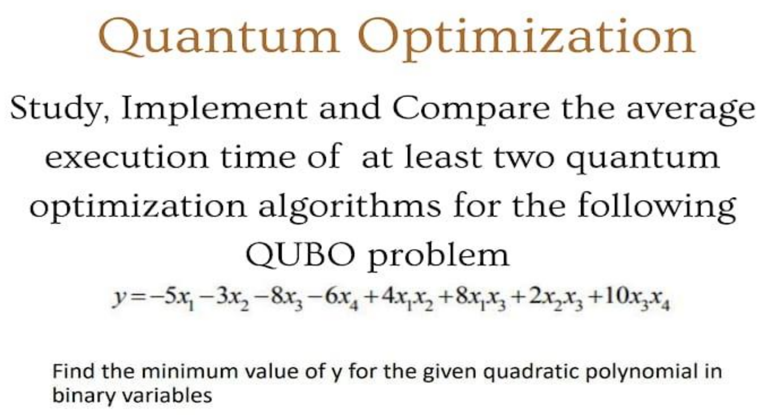
Name: Shisheer S kaushik

Combinatorial Optimization Problem

Combinatorial optimization problem is a problem to find the best solution by solving a minimized optimization problem. If you want to solve a social problem, to formulate this into a combination of binary number **0** and **1** and give some constraint on it.

Problem Statement



Quantum Optimization

Study, Implement and Compare the average execution time of at least two quantum optimization algorithms for the following QUBO problem

$$y=-5x_1-3x_2-8x_3-6x_4+4x_1x_2+8x_1x_3+2x_2x_3+10x_3x_4$$

Find the minimum value of y for the given quadratic polynomial in binary variables

Introduction

Quadratic Unconstrained Binary Optimization (QUBO) problems. Finding the solution to a QUBO is equivalent to finding the ground state of a corresponding Ising Hamiltonian, which is an important problem not only in optimization, but also in quantum chemistry and physics. For this translation, the binary variables taking values in **{0, 1}** are replaced by spin variables taking values in **{−1, +1}**, which allows one to replace the resulting spin variables by Pauli Z matrices, and thus, an Ising Hamiltonian. For more details on this mapping we refer to [1].

Qiskit provides automatic conversion from a suitable QuadraticProgram to an Ising Hamiltonian, which then allows leveraging all the MinimumEigenSolver implementations, such as

- **VQE,**
- **QAOA**, or
- **NumpyMinimumEigensolver** (classical exact method).

Qiskit Optimization provides the MinimumEigenOptimizer class, which wraps the translation to an Ising Hamiltonian (in Qiskit Terra also called Operator), the call to a MinimumEigensolver, and the translation of the results back to an OptimizationResult**.** In the following we first illustrate the conversion from a QuadraticProgram to an Operator and then show how to use the **MinimumEigenOptimizer** with different MinimumEigensolvers to solve a given **QuadraticProgram**. The algorithms in Qiskit automatically try to convert a given problem to the supported problem class if possible, for instance, the **MinimumEigenOptimizer** will automatically translate integer variables to binary variables or add linear equality constraints as a quadratic penalty term to the objective. It should be mentioned that a **QiskitOptimizationError** will be thrown if conversion of a quadratic program with integer variables is attempted.

The circuit depth of **QAOA** potentially has to be increased with the problem size, which might be prohibitive for near-term quantum devices. A possible workaround is Recursive QAOA, as introduced in [2]. Qiskit generalizes this concept to the **RecursiveMinimumEigenOptimizer**, which is introduced at the end of this tutorial.

Defining the Problem

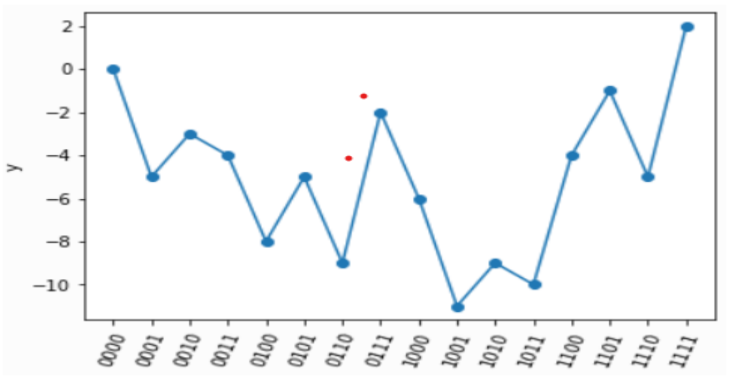
Here we will tackle the simple problem given in Section 2 of Ref 1. We seek the minimum function value, and corresponding configuration of variables, of

$$y = -5x_1 - 3x_2 - 8x_3 - 6x_4 + 4x_1x_2 + 8x_1x_3 + 2x_2x_3 + 10x_3x_4$$

Here the variables xi, i=1, 4 are binary. i.e. they can take the value 0 or 1. Observe that in the linear part (the first 4 terms), all of the variables would ideally be equal to 1 in order to minimise the function. However, the quadratic part (the second 4 terms) encodes penalties for having different pairs of variables equal to 1. Note that in the linear part, we can simply square all of the variables, since xi=x2i for binary variables. We then transform the problem to minimisation of

$$\begin{aligned} y' &= -5x_1^2 - 3x_2^2 - 8x_3^2 - 6x_4^2 + 4x_1x_2 + 8x_1x_3 + 2x_2x_3 + 10x_3x_4 \\ &= \begin{pmatrix} x_4 & x_3 & x_2 & x_1 \end{pmatrix} \begin{pmatrix} -6 & 5 & 0 & 0 \\ 5 & -8 & 1 & 4 \\ 0 & 1 & -3 & 2 \\ 0 & 4 & 2 & -5 \end{pmatrix} \begin{pmatrix} x_4 \\ x_3 \\ x_2 \\ x_1 \end{pmatrix} \end{aligned}$$

We have ordered the variables in the binary strings such that x1 appears as the least significant bit (i.e. on the right of the register), which corresponds to a common convention in the quantum computing community, and in particular for Rigetti’s Pyquil: see here for more information. Let’s plot the function to see what the energy landscape looks like.



The optimal solution is x1=x4=1, x2=x3=0, i.e. the bit string 1001 (the binary representation of the number 9). Its corresponding function value is y=−11.

Converting a QUBO to an Operator

```python
# create a QUBO
qubo = QuadraticProgram()
qubo.binary_var("Z1")
qubo.binary_var("Z2")
qubo.binary_var("Z3")
qubo.binary_var("Z4")
qubo.minimize(linear=[-5, -3, -8, -6], quadratic={("Z1", "Z2"): 4, ("Z1", "Z3"):
 ↪ 8, ("Z2", "Z3"): 2, ("Z3", "Z4"): 10})
print(qubo.prettyprint())
```

```
Problem name:

Minimize
  4*Z1*Z2 + 8*Z1*Z3 + 2*Z2*Z3 + 10*Z3*Z4 - 5*Z1 - 3*Z2 - 8*Z3 - 6*Z4

Subject to
  No constraints


Binary variables (4)
  Z1 Z2 Z3 Z4
```

Next we translate this QUBO into an Ising operator. This results not only in an Operator but also in a constant offset to be taken into account to shift the resulting value. Sometimes a **QuadraticProgram** might also directly be given in the form of an **Operator**. For such cases, Qiskit also provides a translator from an **Operator** back to **QuadraticProgram**, which we illustrate in the following.

```
Problem name:

Minimize
  4*x0*x1 + 8*x0*x2 + 2*x1*x2 + 10*x2*x3 - 5*x0 - 3*x1 - 8*x2 - 6*x3

Subject to
  No constraints

  Binary variables (4)
    x0 x1 x2 x3
```

This translator allows, for instance, one to translate an **Operator** to a **QuadraticProgram** and then solve the problem with other algorithms that are not based on the Ising Hamiltonian representation, such as the **GroverOptimizer**.

Mapping to QAOA

Solving a QUBO with the MinimumEigenOptimizer

1. We start by initializing the MinimumEigensolver we want to use.
2. Then, we use the MinimumEigensolver to create MinimumEigenOptimizer.
3. We first use the MinimumEigenOptimizer based on the classical exact NumPyMinimumEigensolver to get the optimal benchmark solution for this small example.

4. Next we apply the MinimumEigenOptimizer based on QAOA to the same problem.

```
objective function value: -11.0
variable values: Z1=1.0, Z2=0.0, Z3=0.0, Z4=1.0
status: SUCCESS
```

Analysis of Samples

OptimizationResult provides useful information in the form of **SolutionSamples** (here denoted as samples). Each **SolutionSample** contains information about the input values (**x**), the corresponding objective function value (**fval**), the fraction of samples corresponding to that input (**probability**), and the solution status (SUCCESS, FAILURE, INFEASIBLE). Multiple samples corresponding to the same input are consolidated into a single **SolutionSample** (with its probability attribute being the aggregate fraction of samples represented by that **SolutionSample**).

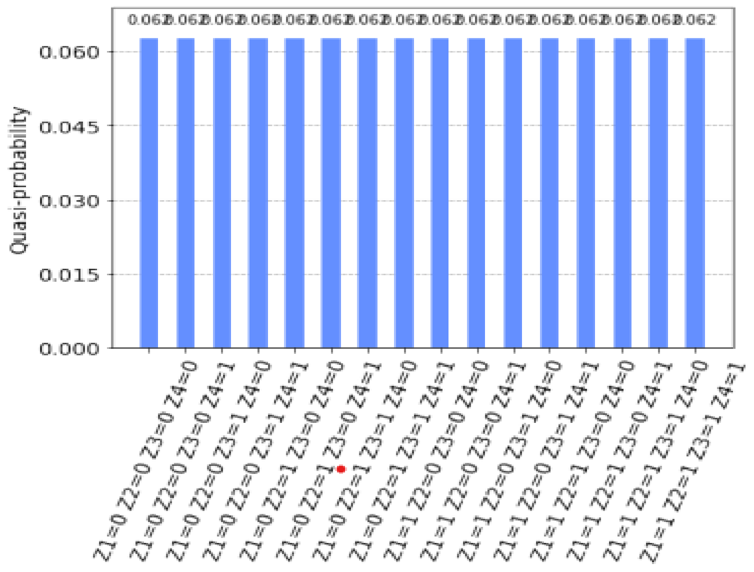
```
SolutionSample(x=array([1., 0., 0., 1.]), fval=-11.0,
probability=0.06249999999999996, status=<OptimizationResultStatus.SUCCESS: 0>)
SolutionSample(x=array([1., 1., 0., 1.]), fval=-10.0,
probability=0.06249999999999996, status=<OptimizationResultStatus.SUCCESS: 0>)
SolutionSample(x=array([0., 1., 1., 0.]), fval=-9.0,
probability=0.06249999999999996, status=<OptimizationResultStatus.SUCCESS: 0>)
SolutionSample(x=array([0., 1., 0., 1.]), fval=-9.0,
probability=0.06249999999999996, status=<OptimizationResultStatus.SUCCESS: 0>)
SolutionSample(x=array([0., 0., 1., 0.]), fval=-8.0,
probability=0.06249999999999996, status=<OptimizationResultStatus.SUCCESS: 0>)
SolutionSample(x=array([0., 0., 0., 1.]), fval=-6.0,
probability=0.06249999999999996, status=<OptimizationResultStatus.SUCCESS: 0>)
SolutionSample(x=array([1., 0., 0., 0.]), fval=-5.0,
probability=0.06249999999999996, status=<OptimizationResultStatus.SUCCESS: 0>)
SolutionSample(x=array([1., 0., 1., 0.]), fval=-5.0,
probability=0.06249999999999996, status=<OptimizationResultStatus.SUCCESS: 0>)
SolutionSample(x=array([0., 1., 1., 1.]), fval=-5.0,
probability=0.06249999999999996, status=<OptimizationResultStatus.SUCCESS: 0>)
SolutionSample(x=array([1., 1., 0., 0.]), fval=-4.0,
probability=0.06249999999999996, status=<OptimizationResultStatus.SUCCESS: 0>)
SolutionSample(x=array([0., 0., 1., 1.]), fval=-4.0,
probability=0.06249999999999996, status=<OptimizationResultStatus.SUCCESS: 0>)
SolutionSample(x=array([0., 1., 0., 0.]), fval=-3.0,
probability=0.06249999999999996, status=<OptimizationResultStatus.SUCCESS: 0>)
SolutionSample(x=array([1., 1., 1., 0.]), fval=-2.0,
probability=0.06249999999999996, status=<OptimizationResultStatus.SUCCESS: 0>)
SolutionSample(x=array([1., 0., 1., 1.]), fval=-1.0,
probability=0.06249999999999996, status=<OptimizationResultStatus.SUCCESS: 0>)
SolutionSample(x=array([0., 0., 0., 0.]), fval=0.0,
probability=0.06249999999999996, status=<OptimizationResultStatus.SUCCESS: 0>)
SolutionSample(x=array([1., 1., 1., 1.]), fval=2.0,
probability=0.06249999999999996, status=<OptimizationResultStatus.SUCCESS: 0>)
```

If we want to obtain a better perspective of the results, statistics is very helpful, both with respect to the objective function values and their respective probabilities. Thus, mean and standard deviation are the very basics for understanding the results.

[mean]: {np.mean (fvals)} → -5.0
[Standard deviation]: {np.std (fvals)} → 3.605551275463989

Finally, despite all the number-crunching, visualization is usually the best early-analysis approach.

```
plot_histogram(samples_for_plot)
```

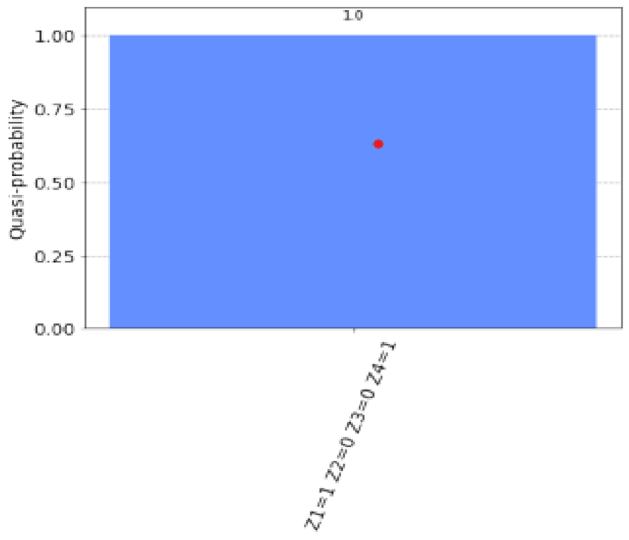


RecursiveMinimumEigenOptimizer

The **RecursiveMinimumEigenOptimizer** takes a **MinimumEigenOptimizer** as input and applies the recursive optimization scheme to reduce the size of the problem one variable at a time. Once the size of the generated intermediate problem is below a given threshold (**min_num_vars**), the **RecursiveMinimumEigenOptimizer** uses another solver (**min_num_vars_optimizer**), e.g., an exact classical solver such as CPLEX or the **MinimumEigenOptimizer** based on the **NumPyMinimumEigensolver**.
In the following, we show how to use the **RecursiveMinimumEigenOptimizer** using the two **MinimumEigenOptimizers** introduced before.

First, we construct the **RecursiveMinimumEigenOptimizer** such that it reduces the problem size from 3 variables to 1 variable and then uses the exact solver for the last variable. Then we call solve to optimize the considered problem.

```
objective function value: -11.0
variable values: Z1=1.0, Z2=0.0, Z3=0.0, Z4=1.0
status: SUCCESS
```



Average Execution Time

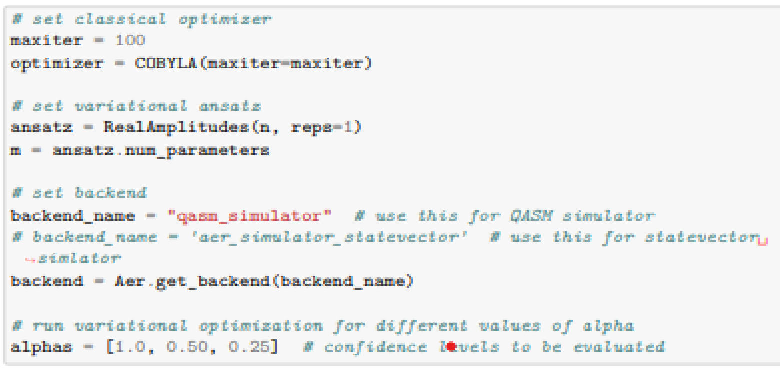
```
timer_ns = end_counter_ns - start_counter_ns
print("Average Execution time:",timer_ns)
```

```
Average Execution time: 6776381187
```

Source Code

Mapping to VQE
Minimum Eigen Optimizer using VQE

This translator allows, for instance, one to translate an Operator to a **QuadraticProgram** and then solve the problem with other algorithms that are not based on the Ising Hamiltonian representation, such as the **GroverOptimizer.**

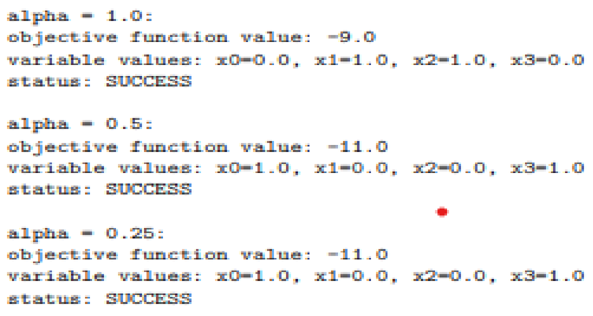


```
# set classical optimizer
maxiter = 100
optimizer = COBYLA(maxiter=maxiter)

# set variational ansatz
ansatz = RealAmplitudes(n, reps=1)
m = ansatz.num_parameters

# set backend
backend_name = "qasm_simulator"  # use this for QASM simulator
# backend_name = 'aer_simulator_statevector'  # use this for statevector␣
↪simlator
backend = Aer.get_backend(backend_name)

# run variational optimization for different values of alpha
alphas = [1.0, 0.50, 0.25]  # confidence levels to be evaluated
```



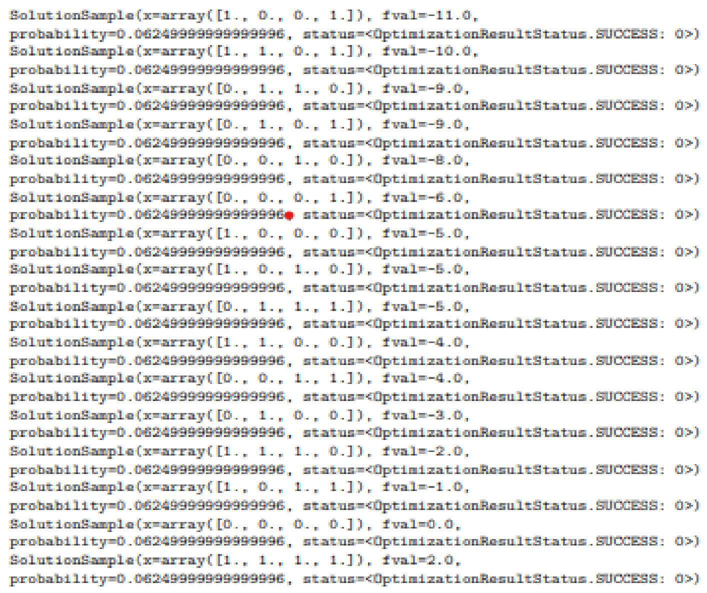
```
alpha = 1.0:
objective function value: -9.0
variable values: x0=0.0, x1=1.0, x2=1.0, x3=0.0
status: SUCCESS

alpha = 0.5:
objective function value: -11.0
variable values: x0=1.0, x1=0.0, x2=0.0, x3=1.0
status: SUCCESS

alpha = 0.25:
objective function value: -11.0
variable values: x0=1.0, x1=0.0, x2=0.0, x3=1.0
status: SUCCESS
```

Analysis of Samples

OptimizationResult provides useful information in the form of **SolutionSamples** (here denoted as samples). Each **SolutionSample** contains information about the input values (**x**), the corresponding objective function value (**fval**), the fraction of samples corresponding to that input (**probability**), and the solution status (SUCCESS, FAILURE, INFEASIBLE). Multiple samples corresponding to the same input are consolidated into a single **SolutionSample** (with its probability attribute being the aggregate fraction of samples represented by that **SolutionSample**).



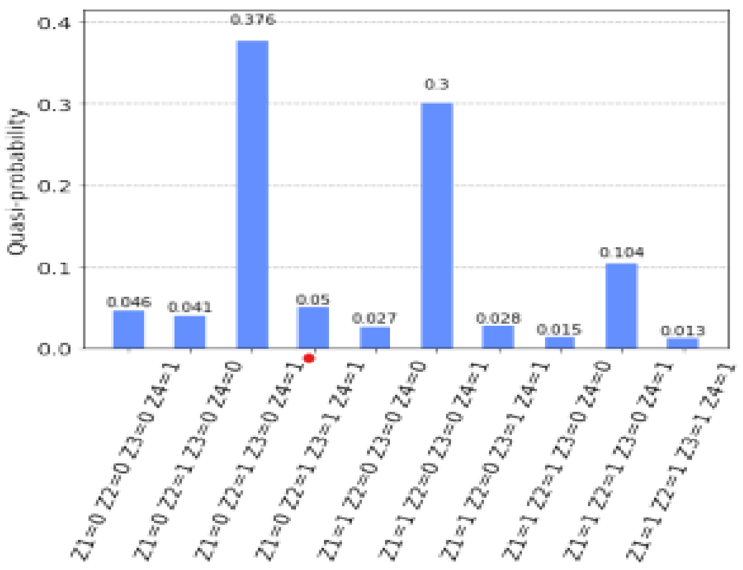
```
SolutionSample(x=array([1., 0., 0., 1.]), fval=-11.0,
probability=0.06249999999999996, status=<OptimizationResultStatus.SUCCESS: 0>)
SolutionSample(x=array([1., 1., 0., 1.]), fval=-10.0,
probability=0.06249999999999996, status=<OptimizationResultStatus.SUCCESS: 0>)
SolutionSample(x=array([0., 1., 1., 0.]), fval=-9.0,
probability=0.06249999999999996, status=<OptimizationResultStatus.SUCCESS: 0>)
SolutionSample(x=array([0., 1., 0., 1.]), fval=-9.0,
probability=0.06249999999999996, status=<OptimizationResultStatus.SUCCESS: 0>)
SolutionSample(x=array([0., 0., 1., 0.]), fval=-8.0,
probability=0.06249999999999996, status=<OptimizationResultStatus.SUCCESS: 0>)
SolutionSample(x=array([0., 0., 0., 1.]), fval=-6.0,
probability=0.06249999999999996, status=<OptimizationResultStatus.SUCCESS: 0>)
SolutionSample(x=array([1., 0., 0., 0.]), fval=-5.0,
probability=0.06249999999999996, status=<OptimizationResultStatus.SUCCESS: 0>)
SolutionSample(x=array([1., 0., 1., 0.]), fval=-5.0,
probability=0.06249999999999996, status=<OptimizationResultStatus.SUCCESS: 0>)
SolutionSample(x=array([0., 1., 1., 1.]), fval=-5.0,
probability=0.06249999999999996, status=<OptimizationResultStatus.SUCCESS: 0>)
SolutionSample(x=array([1., 1., 0., 0.]), fval=-4.0,
probability=0.06249999999999996, status=<OptimizationResultStatus.SUCCESS: 0>)
SolutionSample(x=array([0., 0., 1., 1.]), fval=-4.0,
probability=0.06249999999999996, status=<OptimizationResultStatus.SUCCESS: 0>)
SolutionSample(x=array([0., 1., 0., 0.]), fval=-3.0,
probability=0.06249999999999996, status=<OptimizationResultStatus.SUCCESS: 0>)
SolutionSample(x=array([1., 1., 1., 0.]), fval=-2.0,
probability=0.06249999999999996, status=<OptimizationResultStatus.SUCCESS: 0>)
SolutionSample(x=array([1., 0., 1., 1.]), fval=-1.0,
probability=0.06249999999999996, status=<OptimizationResultStatus.SUCCESS: 0>)
SolutionSample(x=array([0., 0., 0., 0.]), fval=0.0,
probability=0.06249999999999996, status=<OptimizationResultStatus.SUCCESS: 0>)
SolutionSample(x=array([1., 1., 1., 1.]), fval=2.0,
probability=0.06249999999999996, status=<OptimizationResultStatus.SUCCESS: 0>)
```

If we want to obtain a better perspective of the results, statistics is very helpful, both with respect to the objective function values and their respective probabilities. Thus, mean and standard deviation are the very basics for understanding the results.

[mean]: {np.mean (fvals)} → -5.0
[standard deviation]: {np.std (fvals)} → 3.6839419880650364

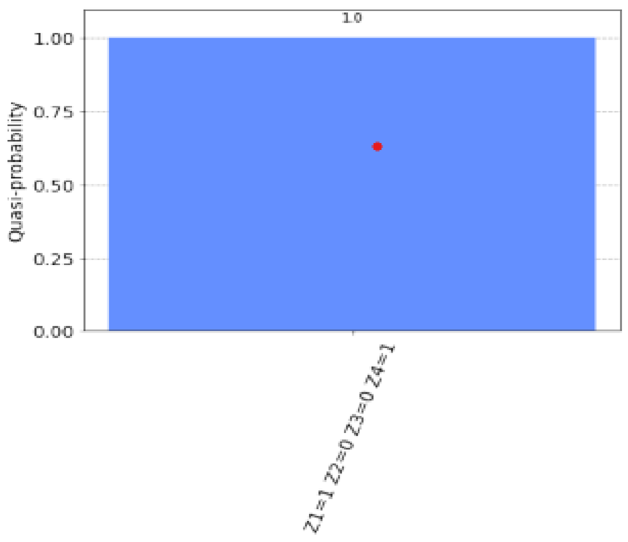
Finally, despite all the number-crunching, visualization is usually the best early-analysis approach.

```
plot_histogram(samples_for_plot)
```



RecursiveMinimumEigenOptimizer

```
objective function value: -11.0
variable values: Z1=1.0, Z2=0.0, Z3=0.0, Z4=1.0
status: SUCCESS
```



Average Execution Time

```
timer_ns = end_counter_ns - start_counter_ns
print("Average Execution time:",timer_ns)
```

```
Average Execution time: 11089415282
```

Source code

References

[1] A. Lucas, Ising formulations of many NP problems, Front. Phys., 12 (2014).

[2] S. ravyi, A. Kliesch, R. Koenig, E. Tang, Obstacles to State Preparation and Variational Optimization from Symmetry Protection, arXiv:1910.08980 (2019)