Subject: Airbus Quantum Computing Challenge / Problem #5 submission report / Aircraft Loading Optimisation

Author: Sorin Moldoveanu

Date: 21.October.2019

COVER PAGE

Contents

Introduction

The purpose is to find solutions (practical demonstration) to aircraft loading optimization by the use of quantum computing. The stated problem is simplified. Few limits are taking into consideration: maximum payload capacity, the space filling, the centre of gravity position and the fuselage shear limits.

Summary

As we already stated we consider a simplified version of the problem. Cargo bay is considered as a box having N – bay spaces. A fisrt approach is to solve the following problem: load the boxes into cargo bay such as loaded mass to be maximized but less than maximum payload AND it must be phisically possible to place boxes into cargo bay. It's a combinatorial problem which will ve translatoed into a binary polynomial optimization problem with constraints. From here an Ising formulation is made and a quantum algorithm is implemented. For quantum implementation we will follow the quantum approximate optimization approach of Farhi, Goldstone, and Gutman (2014), making use of QISKIT, having in mind a quantum computer with gates (also called universal,IBM Q in our situation). Another implementation path is to address a quamtum annealer (Dwawe Systems) – work in progress but this won't be addressed into present article.

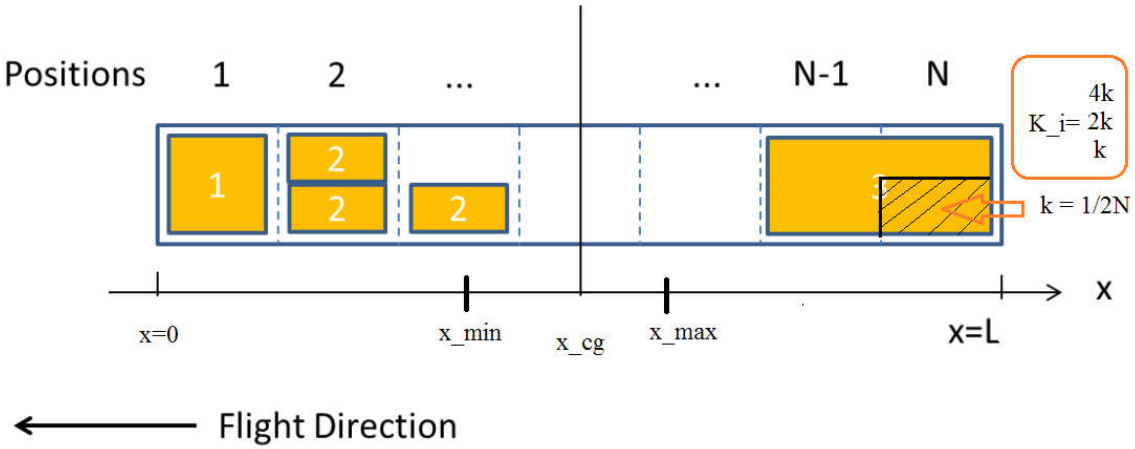
Detalied explanation

The aircraft loading problem is described as a simpliffied version of this kind of problems. Cargo bay is a box with N cargo bay spaces. The cargo boxes (containers) are of 3 types. The optimization target (set of containers which mass has to be maximized within a maximum payload limit) has constraints which are implemented in 3 steps:

Step 1: keep the loaded container mass within the limit and take into accont that it must be physically possible to place the containers on the aircraft

Step 2: place the containers in order to respect the centre of gravity limits

Step 3: the selected set has to respect shear limits of the fuselage



Problem description and formalization; Notations and variables

With respect of container storage, given the N locations, we will consider a new value Ns = 2N (given). We will consider a unit space k as 1/Ns. As a consequence, the space occupied by a given container can be:

$K_i \in \{k, 2k, 4k\}$ where k = 1/N

In this way, the total space occupied by the loaded containers = 1

We will define the following variables:

m_max - maximum payload; scalar; noted Wp into given statement

n - number of mass elements to be loaded - boxes/containers

Ns - number of spaces available for boxes; an unit space is = 1/2 of original_space

M - mass vector n x 1

K - space took by each mass boxes; vector n x 1

x - decision variable vector n x 1; values are 0 or 1

Z - Pauli Z operator

p – penalty

APPROACH #01

**** Step 1 ****

Load the boxes into cargo bay such as loaded mass to be maximized but less that maximum payload AND it must be physically possible to place the boxes in cargo bay. This means

$$max \sum_{i=1}^{n} m_i$$

subject to:

$$\sum_{i=1}^{n} m_i \leqslant m\_max$$

$$\sum_{i=1}^{n} k_i \leqslant 1$$

In order to map the problem for a quantum algorithm, we will introduce a binary decision variable $x \in \{0,1\}^n$ which indicates which mass to pick for x(i) = 1 or not to pick for x(i) = 0.

The constraints are mapped to penalty terms. In vectorial notation:

$(m_{max} - x^T M)^2$

And

$(1 - x^T K)^2$

Those penalty terms are scaled by a parameter p and substracted from the objective function. In this way, the problem is formulated as:

$x^T M - p_0(1 - x^T K)^2 - p_0(m_{max} - x^T M)^2$

The resulting problem can be mapped to an Issing Hamiltonian whose groundstate corresponds to the optimal solution. This can be done using the following assignment:

$x \to (1 - Z_i)/2$

Where Z_i is the Pauli Z operator that has eigenvalues = ± 1

Also note that we have in mind that $x^2 = x$. We will make use of this feature (if case) for our problem processing.

As quantum alghorithm we will implement Variational Qunatum Eigensolver (**VQE**). Also we will use Quantum Approximate Optimization Algorithm (**QAOA**) – as another reference. As a classical reference the exact eigen values can be used.

In brief the **VQE** algorithm can be summarized as follows:

The algorithm computes the expectation value of a Hamialtonian H for an input state $|\psi\rangle$. It uses a parametrized quantum circuit $U(\theta)$ (this is the variational form) to generate trial wave functions $|\psi(\theta)\rangle = U(\theta)|0\rangle$, guided by a classical optimization algorithm that aims to solve $\min_\theta \langle \Psi(\theta) | H | \Psi(\theta) \rangle$ (expression encodes the total energy of a system via its Hamiltonian)

Preparation steps:

Transform the combinatorial problem into a binary polynomial optimization problem with constraints;

Map the resulting problem into an Ising Hamiltonian (H) for variables $\mathbf{z}$ and basis Z ;

Choose the depth of the quantum circuit m. (this can be modified adaptively).

Choose a set of controls θ and make a trial function $|\psi(\boldsymbol{\theta})\rangle$, built using a quantum circuit made of C-Phase gates and single-qubit Y rotations, parameterized by the components of $\boldsymbol{\theta}$.

Algorithm steps:

Evaluate $C(\boldsymbol{\theta}) = \langle \psi(\boldsymbol{\theta}) | H | \psi(\boldsymbol{\theta}) \rangle$ by sampling the outcome of the circuit in the Z-basis and adding the expectation values of the individual Ising terms together. In general, different control points around $\boldsymbol{\theta}$ have to be estimated, depending on the classical optimizer chosen.

Use a classical optimizer to choose a new set of controls.

Continue until $C(\boldsymbol{\theta})$ reaches a minimum, close enough to the solution $\boldsymbol{\theta}^*$.

Use the last $\boldsymbol{\theta}$ to generate a final set of samples from the distribution $\left|\langle z_i \left| \psi(\boldsymbol{\theta}) \right\rangle\right|^2 \ \forall i$ to obtain the answer.

For our implementation we will use the classical optimizer (**COBYLA**) and the variational form (**RY**).

**** Step 2 ****

Add constraints against the mass center positioning (gravity center limits).

Let's define an elemetary length - l:

$$l = L / N$$

Having given the limits of gravity center as x_max and x_min, the constraints for the limits are:

$$l\sum_{i=1}^{N} iXm_i - x_{min}\sum_{i=1}^{N} Xm_i \geqslant 0$$

And

$$-l\sum_{i=1}^{N} iXm_i + x_{max}\sum_{i=1}^{N} Xm_i \geqslant 0$$

where:

$$x \in \{0,1\}^n$$

**** Step 3 ****

Now we have to add the constraints with regards on share limits. Having defined the max for share as S_{max} the constraint is:

$$lx^T M \leqslant S_{max}$$

****NOTE****

The constraints at steps 2 & 3 are to be implemented in the same manner as for the step 1

APPROACH #02

We will take another approach for the objective function, vs: we will consider the cost function related to x_{cg} - the taget centre fo gravity. In this way the function is:

min $\left(l\sum_{i=1}^{N} im_i - m_{max}x_{cg} \right)$

Subject to

$$\sum_{i=1}^{n} m_i \leqslant m_{max}$$

$$\sum_{i=1}^{n} k_i \leqslant 1$$

$$lx^T M \leqslant S_{max}$$

$$l\sum_{i=1}^{N} im_i - x_{min}\sum_{i=1}^{N} m_i \geqslant 0$$

$$-l\sum_{i=1}^{N} im_i + x_{max}\sum_{i=1}^{N} m_i \geqslant 0$$

We will use the same decision variable $x \in \{0,1\}^n$. The constraints are mapped to penalty terms and scaled by a parameter p (as we did for approach#01)

Conclusion

We point that given the use of limited depth of the quantum circuits employed (variational forms), it is hard to discuss the speed-up of the algorithm, as the solution obtained is heuristic in nature. We pointed few references with regards on algorithm performance (evaluations are almost empirical).

An advantage of using implementation on a quantum gates computer (universal) versus an adiabatic/annealing approach is the Hamiltonian does not have to be implemented directly on hardware (in other words the algo is not so limited to the connectivity of the device).

An obvious potential bottleneck on perfirmance is the fact the algorithm requires communications between classical and quantum computers. So, the quantum device and classical computers should be co-located into the same place. At the present time, the code is run via qunatum simultators through QISKIT package.

References

[1] Nielsen M. and Chuang I. "Quantum Computation and Quantum Information", Cambridge University Press (2010)

[2] Qiskit open-source framework (qiskit, qiskit-aqua, qiskit-aer, qiskit-terra) at https://qiskit.org/ and https://github.com/Qiskit

[3] IBMQ at https://quantumexperience.ng.bluemix.net

[4] Patrik J. Coles et al, "Quantum Algorithm Implementation for Beginners", arXiv:1804.03719 (Apr 2018)

[5] Lucas Andrew, "Ising formulations of many NP problems", arXiv"1302.5843 (Jan 2014)

[6] Egger J. Daniel et al, "Qunatum optimization using variational algorithms on near-term quantum devices", arXiv:1710.01022 (Oct 2017)

[7] Edward Fahri et al, "A quantum approximate optimization algorithm", arXiv:1411.4028 (Nov 2014)

[8] Parwadi Moengin, "Polynomial penalty method for solving linear programming problems", International Journal of Applied Mathematics, 40:3, IJAM_40_3_06 (Aug 2010)

[9] Nannicini G, "Performance of hybrid quantum/classical variational heuristics for combinatorial optimization", arXiv:1805.12037 (Dec 2018)

[10] Harrow W Aram et al, "Quantum algorithm for linear systems of equations", arXiv:0811.3171 (Sept 2009)

[11] Lucas Andrew, "Hard combinatorial problems and minor embeddings on lattice graphs", arXiv:1812.01789 (Dec 2018)

[12] Kevin C Young, "Adiabatic quantum optimization with the wrong Hamiltonian", arXiv:1310.0529 (Oct 2013)

[13] Preskill John, "Quantum Computing in the NISQ era and beyond", arXiv:1801.00862 (Jul 2018)

[13] Airbus, "Aircraft Loading Optimization", Airbus Quantum Computing Challenge

[14] Stefan Woerner et al, “Improving Variational Quantum Optimization using CvaR”, arXiv:1907.04769 (July 2019)

[15] Peruzzo Alberto et al, “A variational eigenvalue solver on a quantum processor”, arXiv:1304.3061 (April 2013)

Annexes

Implementation

Qiskit version
qiskit 0.10.1
qiskit-aqua 0.5.0
qiskit-aer 0.2.0
qiskit-terra 0.8.0
qiskit-ibmq-provider 0.2.2
qiskit-ignis 0.1.1

aircargo-pub.ipynb

```
#IMPORT
from qiskit import BasicAer
from qiskit_aqua import QuantumInstance
from qiskit_aqua import Operator, run_algorithm
from qiskit_aqua.input import EnergyInput
from qiskit_aqua.translators.ising import aircargo
from qiskit_aqua.algorithms import VQE, QAOA, ExactEigensolver
from qiskit_aqua.components.optimizers import COBYLA
from qiskit_aqua.components.variational_forms import RY
import numpy as np

#set for real device if case
#device = 'ibmq_16_melbourne'
from qiskit import IBMQ
IBMQ.load_accounts()
#backend = IBMQ.get_backend(device)

# define the problem (instance). Operator instance is created for Issing
# Hamiltonian as it is translated from the problem. The translation module
# is "aircargo.py". We will test with a set of 5 boxes and 4 places in
# cargo #bay.
#
# variables
# m_max - maxumum payload; scalar
# n - number of mass elements to be loaded - boxes
# N - number of spaces available for boxes; an unit space is = 1/2 of
original_space
# M - mass vector n x 1
# K - space took by each mass boxes; vector n x 1
# p - penalty
# e - identity vector
# E - identity matrix
```

```
m_max=6
n=5
N=4  # as defined into the original problem there are 2 storage locations;
here we use a location = 1/2 of original
p = 1/5.0
M=np.array([2.1, 3.2, 0.9, 2.8, 1.2])
#M=np.array([2.1/6, 3.2/6, 0.9/6, 0.8/6, 1.2/6]) # in this case is
reported to m_max; m_max becomes = 1
K=np.array([0.5,0.5,0.25,0.25,0.25])
print('M',M)
print('K',K)
e=np.ones(n)
E = np.matmul(np.asmatrix(e).T, np.asmatrix(e))

qubitOp, offset = aircargo.get_aircargo_qubitops_01(M, K, m_max, p)
#qubitOp, offset = aircargo.get_aircargo_qubitops_fiacco(M, K, m_max, p)
algo_input = EnergyInput(qubitOp)
print('offset=',offset,'qubitOp=', qubitOp)

##########################################
M [2.1 3.2 0.9 2.8 1.2]
K [0.5  0.5  0.25 0.25 0.25]
offset= 14.575500000000003 qubitOp= Representation: paulis, qubits: 5,
size: 15
##########################################

#prepare some printing format
def index_to_selection(i, n):
    s = "{0:b}".format(i).rjust(n)
    x = np.array([1 if s[i]=='1' else 0 for i in reversed(range(n))])
    return x

def print_result(result):
    selection = aircargo.sample_most_likely(result['eigvecs'][0])
    value = aircargo.aircargo_value_01(selection, M, K, m_max, p)
    print('Optimal: selection {}, value {:.4f}'.format(selection, value))

    probabilities = np.abs(result['eigvecs'][0])**2
    i_sorted = reversed(np.argsort(probabilities))
    print('\n----------------- Full result ---------------------')
    print('selection\tvalue\t\tprobability')
    print('---------------------------------------------------')
    for i in i_sorted:
        x = index_to_selection(i, n)
        value = aircargo.aircargo_value_01(x, M, K, m_max, p)
        probability = probabilities[i]
        if value>=0: print('%10s\t%.4f\t\t%.4f' %(x, value, probability))

#exact eigensolver - as reference
exact_eigensolver = ExactEigensolver(qubitOp, k=1)
result = exact_eigensolver.run()
print('\n','Exact_Eigensolver')
print_result(result)
```

```
 exact_eigensolver
Optimal: selection [0 1 0 1 0], value 5.9875

----------------- Full result ---------------------
selection   value      probability
-------------------------------------------------
[0 1 0 1 0] 5.9875           1.0000
[1 1 1 1 1] 6.5595           0.0000
[0 1 1 1 1] 7.2055           0.0000
[0 1 0 0 0] 1.5820           0.0000
[1 1 0 0 0] 5.2020           0.0000
[1 0 1 0 0] 1.1875           0.0000
[0 1 1 0 0] 3.3655           0.0000
[1 1 1 0 0] 6.1795           0.0000
[0 0 0 1 0] 0.6395           0.0000
[1 0 0 1 0] 4.6455           0.0000
[1 1 0 1 0] 7.2055           0.0000
[0 0 1 1 0] 2.5920           0.0000
[1 0 1 1 0] 5.7920           0.0000
[0 1 1 1 0] 6.7380           0.0000
[1 1 1 1 0] 7.1500           0.0000
[1 0 0 0 1] 1.8295           0.0000
[0 1 0 0 1] 3.8755           0.0000
[1 1 0 0 1] 6.4375           0.0000
[1 0 1 0 1] 3.5520           0.0000
[0 1 1 0 1] 5.2020           0.0000
[1 1 1 0 1] 6.9580           0.0000
[0 0 0 1 1] 3.1500           0.0000
[1 0 0 1 1] 6.0980           0.0000
[0 1 0 1 1] 6.9120           0.0000
[1 1 0 1 1] 7.0720           0.0000
[0 0 1 1 1] 4.6455           0.0000
[1 0 1 1 1] 6.7875           0.0000

#VQE
backend = BasicAer.get_backend('statevector_simulator')
seed = 50

cobyla = COBYLA()
cobyla.set_options(maxiter=250)
ry = RY(qubitOp.num_qubits, depth=3, entanglement='full')
#num_of_parameters = d x q x (q+1)/2 +q where d = circuit depth and q =
numb of qubits
#depth example: d=1 means one step quantum algo - can be expressed as a
unitary operator
vqe = VQE(qubitOp, ry, cobyla, 'matrix')
vqe.random_seed = seed

quantum_instance = QuantumInstance(backend=backend, seed=seed,
seed_mapper=seed)

result = vqe.run(quantum_instance)
print('\n','VQE')
print_result(result)
```

```
#in order to know circut width and depth: how many qubits are required
(circuit width)
# or how large the maximum number of gates applied to a single qubit
(circuit depth) is

#print("circuit_width", result['circuit_info']['width'])
#print("circuit_depth", result['circuit_info']['depth'])

############################################################
VQE
Optimal: selection [1 1 0 0 0], value 5.2020

----------------- Full result ---------------------
selection   value      probability
-------------------------------------------------
[1 1 0 0 0] 5.2020           0.9987
[1 0 1 0 0] 1.1875           0.0004
[1 0 0 0 1] 1.8295           0.0002
[1 0 1 0 1] 3.5520           0.0002
[1 1 0 1 0] 7.2055           0.0002
[1 1 1 0 0] 6.1795           0.0001
[1 1 1 0 1] 6.9580           0.0001
[0 1 1 0 1] 5.2020           0.0000
[1 0 0 1 0] 4.6455           0.0000
[1 1 0 0 1] 6.4375           0.0000
[0 1 1 0 0] 3.3655           0.0000
[0 1 0 0 0] 1.5820           0.0000
[0 1 0 0 1] 3.8755           0.0000
[1 1 1 1 0] 7.1500           0.0000
[1 1 0 1 1] 7.0720           0.0000
[1 0 1 1 1] 6.7875           0.0000
[1 0 0 1 1] 6.0980           0.0000
[1 0 1 1 0] 5.7920           0.0000
[1 1 1 1 1] 6.5595           0.0000
[0 0 1 1 1] 4.6455           0.0000
[0 0 0 1 1] 3.1500           0.0000
[0 1 1 1 1] 7.2055           0.0000
[0 0 1 1 0] 2.5920           0.0000
[0 1 1 1 0] 6.7380           0.0000
[0 1 0 1 1] 6.9120           0.0000
[0 1 0 1 0] 5.9875           0.0000
[0 0 0 1 0] 0.6395           0.0000
############################################################

#QAOA
backend = BasicAer.get_backend('statevector_simulator')
seed = 50

cobyla = COBYLA()
cobyla.set_options(maxiter=250)
qaoa = QAOA(qubitOp, cobyla, 3, 'matrix')
qaoa.random_seed = seed
```

```
quantum_instance = QuantumInstance(backend=backend, seed=seed,
seed_mapper=seed)

result = qaoa.run(quantum_instance)
print('\n','QAOA')
print_result(result)

#############################################################
QAOA
Optimal: selection [0 1 0 1 0], value 5.9875

----------------- Full result ---------------------
selection   value      probability
-------------------------------------------------
[0 1 0 1 0] 5.9875           0.1073
[1 1 0 0 0] 5.2020           0.0946
[1 0 0 1 0] 4.6455           0.0860
[0 1 0 0 1] 3.8755           0.0798
[0 1 1 0 0] 3.3655           0.0763
[0 0 0 1 1] 3.1500           0.0728
[0 0 1 1 0] 2.5920           0.0697
[1 0 0 0 1] 1.8295           0.0649
[1 0 1 0 0] 1.1875           0.0622
[1 1 0 1 0] 7.2055           0.0406
[0 1 0 1 1] 6.9120           0.0296
[0 1 1 1 0] 6.7380           0.0267
[1 1 0 0 1] 6.4375           0.0242
[1 1 1 0 0] 6.1795           0.0218
[1 0 0 1 1] 6.0980           0.0211
[1 0 1 1 0] 5.7920           0.0189
[0 1 1 0 1] 5.2020           0.0153
[0 0 1 1 1] 4.6455           0.0131
[1 0 1 0 1] 3.5520           0.0103
[0 1 0 0 0] 1.5820           0.0018
[0 0 0 1 0] 0.6395           0.0012
[1 1 1 1 1] 6.5595           0.0005
[1 1 0 1 1] 7.0720           0.0003
[1 1 1 1 0] 7.1500           0.0002
[0 1 1 1 1] 7.2055           0.0001
[1 1 1 0 1] 6.9580           0.0001
[1 0 1 1 1] 6.7875           0.0001
#############################################################
```

############

aircargo.py

############

```
# Convert airbus_05 air_cargo optimization instances into Pauli list

from collections import OrderedDict

import numpy as np
from qiskit.quantum_info import Pauli
```

```
from qiskit_aqua import Operator

from sklearn.datasets import make_spd_matrix


def get_aircargo_qubitops(M, K, m_max, p):

    # get problem dimension
    n = len(M)
    e = np.ones(n)
    E = np.matmul(np.asmatrix(e).T, np.asmatrix(e))

    # model is -/-/- => -xM -p*(xK-E)**2 - p*(xM-Max)**2
    # map problem to Ising model
    # offset - scalar / mu_z - vector / sigma_z - matrix
    offset = -np.dot(e,M)/2 - p/4*np.dot(K,K) + p*np.dot(e,K) -
p*np.dot(e,e) - p/4*np.dot(M,M) + p*m_max*np.dot(e,M) - p*m_max**2
    mu_z = M/2 + p/2*np.dot(K,K) - p*K + p/2*np.dot(M,M) - p*m_max*M
    sigma_z = -p/4*np.dot(E,np.dot(K,K)) - p/4*np.dot(E,np.dot(M,M))

    # construct operator
    pauli_list = []
    for i in range(n):
        i_ = i
        # i_ = n-i-1
        if np.abs(mu_z[i_]) > 1e-6:
            xp = np.zeros(n, dtype=np.bool)
            zp = np.zeros(n, dtype=np.bool)
            zp[i_] = True
            pauli_list.append([mu_z[i_], Pauli(zp, xp)])
        for j in range(i):
            j_ = j
            # j_ = n-j-1
            if np.abs(sigma_z[i_, j_]) > 1e-6:
                xp = np.zeros(n, dtype=np.bool)
                zp = np.zeros(n, dtype=np.bool)
                zp[i_] = True
                zp[j_] = True
                pauli_list.append([2*sigma_z[i_, j_], Pauli(zp, xp)])
        offset += sigma_z[i_, i_]

    return Operator(paulis=pauli_list), offset

def get_aircargo_qubitops_01(M, K, m_max, p):

    # get problem dimension
    n = len(M)
    e = np.ones(n)
    E = np.matmul(np.asmatrix(e).T, np.asmatrix(e))
    pk = p

    # map problem to Ising model
    # model is +/-/- => xM - p*(E-xK)**2 - p*(Max-xM)**2
```

```
    offset = np.dot(e,M)/2 - pk/4*np.dot(K,K) + pk*np.dot(e,K) -
p*np.dot(e,e) - p/4*np.dot(M,M) + p*m_max*np.dot(e,M) - p*m_max**2
    mu_z = -M/2 - pk/2*np.dot(K,K) + pk*K - p/2*np.dot(M,M) + p*m_max*M
    sigma_z = pk/4*np.dot(E,np.dot(K,K)) + p/4*np.dot(E,np.dot(M,M))

    # construct operator
    pauli_list = []
    for i in range(n):
        i_ = i
        # i_ = n-i-1
        if np.abs(mu_z[i_]) > 1e-6:
            xp = np.zeros(n, dtype=np.bool)
            zp = np.zeros(n, dtype=np.bool)
            zp[i_] = True
            pauli_list.append([mu_z[i_], Pauli(zp, xp)])
        for j in range(i):
            j_ = j
            # j_ = n-j-1
            if np.abs(sigma_z[i_, j_]) > 1e-6:
                xp = np.zeros(n, dtype=np.bool)
                zp = np.zeros(n, dtype=np.bool)
                zp[i_] = True
                zp[j_] = True
                pauli_list.append([2*sigma_z[i_, j_], Pauli(zp, xp)])
        offset += sigma_z[i_, i_]

    return Operator(paulis=pauli_list), offset


def get_aircargo_qubitops_02(M, K, m_max, p):

    # get problem dimension
    n = len(M)
    e = np.ones(n)
    E = np.matmul(np.asmatrix(e).T, np.asmatrix(e))
    pk = p

    # map problem to Ising model
    # model is +/-/- => xM**2 - p*(E-xK)**2 - p*(Max-xM)**2
    offset = np.dot(e,M)**2/4 - pk/4*np.dot(K,K) + pk*np.dot(e,K) -
p*np.dot(e,e) - p/4*np.dot(M,M) + p*m_max*np.dot(e,M) - p*m_max**2
    mu_z = -M**2/2 - pk/2*np.dot(K,K) + pk*K - p/2*np.dot(M,M) + p*m_max*M
    sigma_z = pk/4*np.dot(E,np.dot(K,K)) + p/4*np.dot(E,np.dot(M,M)) +
np.dot(E,np.dot(M,M))/4

    # construct operator
    pauli_list = []
    for i in range(n):
        i_ = i
        # i_ = n-i-1
        if np.abs(mu_z[i_]) > 1e-6:
            xp = np.zeros(n, dtype=np.bool)
            zp = np.zeros(n, dtype=np.bool)
            zp[i_] = True
```

```
            pauli_list.append([mu_z[i_], Pauli(zp, xp)])
        for j in range(i):
            j_ = j
            # j_ = n-j-1
            if np.abs(sigma_z[i_, j_]) > 1e-6:
                xp = np.zeros(n, dtype=np.bool)
                zp = np.zeros(n, dtype=np.bool)
                zp[i_] = True
                zp[j_] = True
                pauli_list.append([0.5*sigma_z[i_, j_], Pauli(zp, xp)])
        offset += sigma_z[i_, i_]

    return Operator(paulis=pauli_list), offset

def get_aircargo_qubitops_fiacco(M, K, m_max, p):

    # get problem dimension
    n = len(M)
    e = np.ones(n)
    E = np.matmul(np.asmatrix(e).T, np.asmatrix(e))
    psq=p**(1/2.0)

    # map problem to Ising model
    # model is +/-/- => -xM + (E-xK)**2/sqrt(p) + p*1/(Max-xM)
    offset = -np.dot(e,M)/2 + psq/4*np.dot(K,K) - psq*np.dot(e,K) +
psq*np.dot(e,e) + p*(1-m_max+np.dot(e,M)/2)
    mu_z = +M/2 - psq/2*np.dot(K,K) + psq*K - p/2*M
    sigma_z = psq/4*np.dot(E,np.dot(K,K))

    # construct operator
    pauli_list = []
    for i in range(n):
        i_ = i
        # i_ = n-i-1
        if np.abs(mu_z[i_]) > 1e-6:
            xp = np.zeros(n, dtype=np.bool)
            zp = np.zeros(n, dtype=np.bool)
            zp[i_] = True
            pauli_list.append([mu_z[i_], Pauli(zp, xp)])
        for j in range(i):
            j_ = j
            # j_ = n-j-1
            if np.abs(sigma_z[i_, j_]) > 1e-6:
                xp = np.zeros(n, dtype=np.bool)
                zp = np.zeros(n, dtype=np.bool)
                zp[i_] = True
                zp[j_] = True
                pauli_list.append([2*sigma_z[i_, j_], Pauli(zp, xp)])
        offset += sigma_z[i_, i_]

    return Operator(paulis=pauli_list), offset


def aircargo_value(x, M, K, m_max, p):
```

```
    return -np.dot(x,M) - p*pow(np.dot(x,K)-1, 2) - p*pow(np.dot(x,M)-
m_max,2)

def aircargo_value_01(x, M, K, m_max, p):
    return np.dot(x,M) - p*pow(1-np.dot(x,K), 2) - p*pow(m_max-
np.dot(x,M),2)

def aircargo_value_02(x, M, K, m_max, p):
    return pow(np.dot(x,M),2) - p*pow(1-np.dot(x,K), 2) - p*pow(m_max-
np.dot(x,M),2)


def sample_most_likely(state_vector):
    """Compute the most likely binary string from state vector.

    Args:
        state_vector (numpy.ndarray or dict): state vector or counts.

    Returns:
        numpy.ndarray: binary string as numpy.ndarray of ints.
    """
    if isinstance(state_vector, dict) or isinstance(state_vector,
OrderedDict):
        # get the binary string with the largest count
        binary_string = sorted(state_vector.items(), key=lambda kv:
kv[1])[-1][0]
        x = np.asarray([int(y) for y in reversed(list(binary_string))])
        return x
    else:
        n = int(np.log2(state_vector.shape[0]))
        k = np.argmax(np.abs(state_vector))
        x = np.zeros(n, dtype=int)
        for i in range(n):
            x[i] = k % 2
            k >>= 1
        return x
```