Subject: Airbus Quantum Computing Challenge / Problem #1 submission report / Aircraft Climb Optimisation

Author: Sorin Moldoveanu

Date: 22.October.2019

COVER PAGE

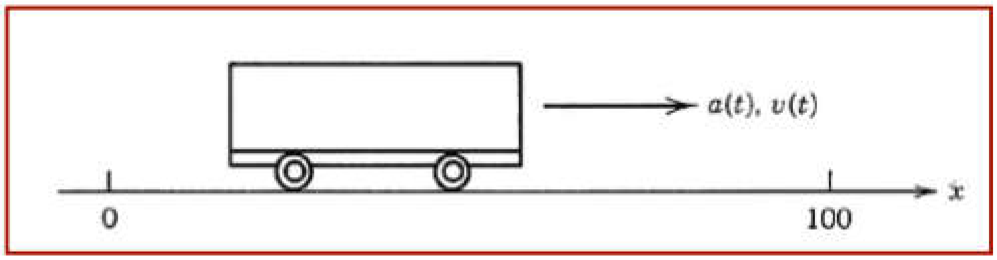
Contents

Introduction

The purpose is to find solutions (practical demonstration) to aircraft mission optimization by the use of quantum computing, focusing on the optimization of the climb trajectory. The stated problem is complex. Due to degree of complexity we will try to prove using a much more simpler problem - cart on a track .

Summary

As we already stated we consider a more simpler control problem: cart on a track – simple linear form.



We will try to find out the minimum (constant) force applied to a cart moving free on a track. In other words, use as little force as possible.

Example: we want to move the cart 100m in 10s. The initial velocity v is 0, the final velocity is free. The mass of the cart m is constant. The only control is the force f applied, namely the acceleration a.

We consider 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

In order to describe what happens, equations are:

$$\dot{x} = v$$

$$\dot{v} = f / m = a$$

Notations:

$$x = x_1$$

$$v = x_2$$

$$a = u = k = f / m$$

The function to minimize:

$$J = q(x_{1f} - 100)^2 + r\int_0^{t_f} a^2 dt$$

Working a little on J, considering a=$const$, the function comes to:

$$J = q(50k - 100)^2 + 10rk^2$$

In order to solve on a quantum computer, we put the function in a binary form, with a binary decision variable $x \in \{0,1\}^n$ which indicate which control value to pick:

$$J = q(50x^T K - 100)^2 + 10rx^T K^2$$

subject to restriction to pick one control value only:

$$x^T K = 1$$

In the end, the objective is:

$$min\left(q(50x^T K - 100)^2 + 10rx^T K^2 + p(1 - x^T K)^2\right)$$

where K is a vector of possible control(acceleration) values (for this case considering a sigle time interval) and q, r, p are penalty terms (chose in such way in order to satisfy each objective)

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:

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

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.

The module for value and Pauli operator calculus is cargo.py

NOTE:

- The analytical solutions are: K=0 (obvoius) and K=2 for $q/r \rightarrow \infty$

As quantum alghorithm we will implement Variational Qunatum Eigensolver (**VQE**). Also we will use Quantum Approximate Optimization Algorithm (**QAOA**) – as other 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

$\left|\psi(\theta)\right\rangle = U(\theta)\,|\,0\rangle$, guided by a classical optimization algorithm that aims to solve $\min_\theta \left\langle \Psi(\theta)\,|\,H\,|\,\Psi(\theta)\right\rangle$ (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 $\left|\psi(\boldsymbol{\theta})\right\rangle$, 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 $C(\boldsymbol{\theta}) = \langle\psi(\boldsymbol{\theta})\big|\,H\,\big|\psi(\boldsymbol{\theta})\rangle$ 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 $\left|\langle z_i\,\big|\psi(\boldsymbol{\theta})\rangle\right|^2 \ \forall i$ to obtain the answer.

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

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] Roert F. Stengel, “Optimal Control and Estimation”, Courier Corporation (1994)

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

[12] Airbus, "Aircraft Climb Optimization", Airbus Quantum Computing Challenge

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

[14] 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

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

# cartonatrack-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 cart
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
#device = 'ibmq_16_melbourne'
from qiskit import IBMQ
IBMQ.load_accounts()
#backend = IBMQ.get_backend(device)

# define the problem
# variables
# n - number of acceleration elements to be taken into account
# K - acceleration vector; vector n x 1
# q - penalty
# p - penalty
# r - penalty

n = 5
q = 10**6
r = 1
p = 10
K=np.array([1.5,1.7,2,2.1,2.4])
e=np.ones(n)
E = np.matmul(np.asmatrix(e).T, np.asmatrix(e))
```

```
qubitOp, offset = cart.get_cart_qubitops(K, q, r, p)
algo_input = EnergyInput(qubitOp)
#print(offset, qubitOp)

#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 = cart.sample_most_likely(result['eigvecs'][0])
    #print('selection',selection)
    value = cart.cart_value(selection, K, q, r, p)
    print('\nvalue:',value)
    print('Optimal: selection {}, value {:.4f}'.format(selection, value))
    #print(selection,value)

    probabilities = np.abs(result['eigvecs'][0])**2
    i_sorted = reversed(np.argsort(probabilities))
    #i_sorted = reversed(np.argsort(value))
    print('\n----------------- Full result ---------------------')
    print('selection\tvalue\t\tprobability')
    print('---------------------------------------------------')
    for i in i_sorted:
        x = index_to_selection(i, n)
        #print('\nx=',x)
        value = cart.cart_value(x, K, q, r, p)
        probability = probabilities[i]
        print('%10s\t%.4f\t\t%.4f' %(x, value, probability))
        #if value>=0: print('%10s\t%.4f\t\t%.4f' %(x, value, probability))
        #if len(np.argwhere(x>0))==1: 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(result)
#print(result['eigvecs'][0])
print('\n','EXACT_eigensolver')
print_result(result)

************************************************
EXACT_eigensolver

value: -10000000010.0
Optimal: selection [0 0 0 0 0], value -10000000010.0000

----------------- Full result ---------------------
selection   value       probability
---------------------------------------------------
[0 0 0 0 0] -10000000010.0000       1.0000
[0 1 1 1 1] -96100000347.8000       0.0000
[1 0 0 0 0] -624999980.0000     0.0000
[0 1 0 0 0] -224999976.0000     0.0000
[1 1 0 0 0] -3599999997.0000        0.0000
```

```
[0 0 1 0 0] 30.0000     0.0000
[1 0 1 0 0] -5625000000.0000        0.0000
[0 1 1 0 0] -7225000004.0000        0.0000
[1 1 1 0 0] -25600000085.0000       0.0000
[0 0 0 1 0] -24999968.0000      0.0000
[1 0 0 1 0] -6400000001.0000        0.0000
[0 1 0 1 0] -8100000005.4000        0.0000
[1 1 0 1 0] -27225000089.4000       0.0000
[0 0 1 1 0] -11025000012.0000       0.0000
[1 0 1 1 0] -32400000105.0000       0.0000
[0 1 1 1 0] -36100000117.4000       0.0000
[1 1 1 1 1] -148225000563.8000      0.0000
[0 0 0 0 1] -399999962.0000     0.0000
[1 0 0 0 1] -9025000004.0000        0.0000
[0 1 0 0 1] -11025000009.6000       0.0000
[1 1 0 0 1] -32400000102.6000       0.0000
[0 0 1 0 1] -14400000018.0000       0.0000
[1 0 1 0 1] -38025000120.0000       0.0000
[0 1 1 0 1] -42025000133.6000       0.0000
[1 1 1 0 1] -78400000286.6000       0.0000
[0 0 0 1 1] -15625000020.8000       0.0000
[1 0 0 1 1] -40000000125.8000       0.0000
[0 1 0 1 1] -44100000139.8000       0.0000
[1 1 0 1 1] -81225000295.8000       0.0000
[0 0 1 1 1] -50625000160.8000       0.0000
[1 0 1 1 1] -90000000325.8000       0.0000
[1 1 1 1 0] -70225000261.4000       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')
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)

*********************************************************
VQE

value: -10000000010.0
Optimal: selection [0 0 0 0 0], value -10000000010.0000

------------------ Full result ---------------------
selection   value       probability
---------------------------------------------------
[0 0 0 0 0] -10000000010.0000       0.7741
[1 1 1 1 1] -148225000563.8000      0.2107
```

```
[0 0 1 0 0] 30.0000     0.0047
[0 1 1 1 1] -96100000347.8000       0.0033
[1 1 0 1 1] -81225000295.8000       0.0031
[1 0 0 0 0] -624999980.0000     0.0014
[1 0 1 0 0] -5625000000.0000        0.0011
[0 1 0 0 0] -224999976.0000     0.0007
[0 0 1 1 1] -50625000160.8000       0.0002
[1 0 1 1 1] -90000000325.8000       0.0001
[0 1 0 1 1] -44100000139.8000       0.0001
[1 0 0 1 1] -40000000125.8000       0.0001
[0 0 0 1 0] -24999968.0000      0.0001
[1 1 1 0 1] -78400000286.6000       0.0001
[1 1 1 0 0] -25600000085.0000       0.0000
[0 0 0 1 1] -15625000020.8000       0.0000
[0 1 1 0 1] -42025000133.6000       0.0000
[1 1 1 1 0] -70225000261.4000       0.0000
[1 1 0 0 0] -3599999997.0000        0.0000
[0 1 0 0 1] -11025000009.6000       0.0000
[1 0 0 1 0] -6400000001.0000        0.0000
[0 0 0 0 1] -399999962.0000     0.0000
[1 0 1 1 0] -32400000105.0000       0.0000
[1 0 0 0 1] -9025000004.0000        0.0000
[0 1 1 0 0] -7225000004.0000        0.0000
[0 0 1 0 1] -14400000018.0000       0.0000
[0 1 0 1 0] -8100000005.4000        0.0000
[1 1 0 0 1] -32400000102.6000       0.0000
[0 0 1 1 0] -11025000012.0000       0.0000
[0 1 1 1 0] -36100000117.4000       0.0000
[1 1 0 1 0] -27225000089.4000       0.0000
[1 0 1 0 1] -38025000120.0000       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

value: 5062944.2
Optimal: selection [0 0 1 1 1], value 5062944.2000

----------------- Full result ---------------------
selection   value       probability
---------------------------------------------------
[0 0 1 1 1] 5062944.2000        0.1588
```

```
[0 0 0 1 1] 1562724.2000        0.1059
[0 0 1 0 0] 50.0000     0.0864
[1 1 1 0 0] 7023032.4000        0.0812
[0 1 0 1 0] 810151.4000     0.0697
[1 0 0 1 0] 640134.2000     0.0649
[0 0 1 0 1] 1440213.2000        0.0580
[0 0 0 0 0] 1000010.0000        0.0455
[1 0 1 1 1] 9000654.2000        0.0390
[0 1 0 0 1] 1102682.6000        0.0323
[0 1 0 0 0] 22533.8000      0.0281
[1 0 0 1 1] 4000374.2000        0.0276
[1 1 1 0 0] 2560267.8000        0.0227
[1 1 1 1 1] 14823450.0000       0.0217
[1 1 0 1 0] 2722780.4000        0.0202
[0 0 1 1 0] 1102680.2000        0.0168
[1 0 1 0 0] 562625.0000     0.0148
[1 1 1 0 1] 7840584.6000        0.0138
[1 0 0 0 1] 902664.2000     0.0128
[0 1 1 1 0] 3610343.4000        0.0127
[1 1 0 1 1] 8123102.0000        0.0102
[1 0 1 0 1] 3802860.2000        0.0095
[0 1 1 0 1] 4202886.6000        0.0091
[0 0 0 0 1] 40077.2000      0.0089
[1 0 1 1 0] 3240318.2000        0.0085
[0 1 1 1 1] 9610689.0000        0.0083
[0 1 1 0 0] 722641.8000     0.0073
[1 0 0 0 0] 62525.0000      0.0039
[0 1 0 1 1] 4410401.0000        0.0013
[0 0 0 1 0] 2556.2000       0.0001
[1 1 0 0 1] 3240320.6000        0.0000
[1 1 0 0 0] 360099.8000     0.0000
******************************************************
```

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

# cart.py

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

```
# Convert cart on a track 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_cart_qubitops(K, q, r, p):
    # get problem dimension
    n = len(K)
    e = np.ones(n)
    E = np.matmul(np.asmatrix(e).T, np.asmatrix(e))
    t = 10
    #p = 10000

    # map problem to Ising model
    # model is => x*a - r*(100 - x*a*t^2)
    #offset = np.dot(e,K)/2 + np.dot(e,K)**2*r*t**4/4 - 10**4*r -
10**2*np.dot(e,K)*t**2
    #second model is => q*(50*x*K-100)^2 + 10*r*x*K^2 + p*(x*K-1)^2
    #offset = (50**2*q/4+5*r+p/2)*np.dot(e,K)**2-
(100**2*q/2+p)*np.dot(e,K)+100**2*q+p
    #model : -q*(50*np.dot(x,K)-100)**2 + 10*r*np.dot(x^2,np.power(K,2)) -
p*(np.dot(x,K)-1)**2
    offset = (-50**2*q/4+5*r-p/2)*np.dot(e,K)**2+(100**2*q/2+p)*np.dot(e,K)-
100**2*q-p

    #mu_z = -K/2 - r*K**2*t**4/2 + 10**2*K*t**2
    #mu_z = (100**2*q/2+p)*K-(50**2*q/2+5*r+p)*np.power(K,2)
    mu_z = -(100**2*q/2+p)*K-(-50**2*q/2+10*r-p)*np.power(K,2)

    #sigma_z = np.dot(E,np.dot(K,K))*r*t**4/4
    #sigma_z =
(50**2*q/4+p/2)*np.matmul(np.asmatrix(np.power(K,2)).T,np.asmatrix(np.power(K
,2)))
    sigma_z = (5*r-50**2*q/2-
p/2)*np.matmul(np.asmatrix(np.power(K,2)).T,np.asmatrix(np.power(K,2)))

    # 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

    # 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 cart_value(x, K, q, r, p):
    n=len(K)
    e=np.ones(n)
    #p=10
    #return np.dot(x,K) - r*(100-100*np.dot(x,K))**2
    #return q*(50*np.dot(x,K)-100)**2 + 10*r*np.dot(x,np.power(K,2)) +
p*(np.dot(x,K)-1)**2
    return -q*(50*np.dot(x,K)-100)**2 + 10*r*np.dot(x,np.power(K,2)) -
p*(np.dot(x,K)-1)**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
```