Quantum Inspire Examples

QuTech

Apr 14, 2023

CONTENTS

Hello, and welcome to the Quantum Inspire Examples.

The examples mainly consist of Jupyter notebooks and python scripts.

CHAPTER

ONE

DOCUMENTATION

1.1 Introduction

Welcome to the Quantum Inspire Examples. This introduction will shortly introduce the repository, and it will guide you through the structure, installation process and how to contribute. We look forward to working with you!

The Quantum Inspire Examples consist of a number of Jupyter notebooks and python scripts with a diverse set of Quantum algorithms that illustrate the possibilities of the Quantum Inspire platform to run more complex algorithms. The Quantum Inspire examples make use of:

- An API for the Quantum Inspire platform (the Quantum Inspire SDK);
- Backends for:
  - the ProjectQ SDK;
  - the Qiskit SDK.

For more information on Quantum Inspire see https://www.quantum-inspire.com/. Detailed information can be found in the Quantum Inspire knowledge base.

Quantum Inspire is developed by QuTech QuTech is an advanced research center based in Delft, the Netherlands, for quantum computing and quantum internet. It is a collaboration founded by the Delft University of Technology (TU Delft) and the Netherlands Organisation for Applied Scientific Research (TNO).

1.1.1 Installing from source

The source for the Quantum Inspire examples can be found at Github. For the default installation execute:

```
git clone https://github.com/QuTech-Delft/quantum-inspire-examples
cd quantum-inspire-examples
git submodule update --init
pip install .
```

This will install everything necessary to run the examples, the Quantum Inspire SDK including the Qiskit and ProjectQ packages.

1.1.2 Installing for generating documentation

To install the necessary packages to perform documentation activities:

```
pip install .[rtd]
```

The documentation generation process is dependent on pandoc. When you want to generate the documentation and pandoc is not yet installed on your system navigate to Pandoc and follow the instructions found there to install pandoc. To build the 'readthedocs' documentation do:

```
cd docs
make html
```

The documentation is then build in 'docs/_build/html'.

1.1.3 Running

For example usage see the python scripts in the `docs/examples/` directory and Jupyter notebooks in the `docs/notebooks/` directory when installed from source.

For example, to run the ProjectQ example notebook after installing from source:

```
cd docs/notebooks
jupyter notebook example_projectq.ipynb
```

or when you want to choose which example notebook to run from the browser do:

```
jupyter notebook --notebook-dir="docs/notebooks"
```

and select a Jupyter notebook (file with extension `ipynb`) to run from one of the directories.

1.1.4 Configure your token credentials for Quantum Inspire

To make use of Quantum Inspire requires you to register and create an account. To prevent submitting your credentials with each example you can make use of token authentication.

1. Create a Quantum Inspire account at https://www.quantum-inspire.com/ if you do not already have one.
2. Get an API token from the Quantum Inspire website https://www.quantum-inspire.com/account.
3. With your API token run:

```
from quantuminspire.credentials import save_account
save_account('YOUR_API_TOKEN')
```

After calling `save_account`, your credentials will be stored on disk and token authentication is done automatically in many of the examples.

1.2 Example notebooks

1.2.1 Some basic examples

Example to use ProjectQ to run algorithms on Quantum Inspire

For more information on Quantum Inspire, see https://www.quantum-inspire.com/. For more information on ProjectQ, see https://github.com/ProjectQ-Framework/ProjectQ.

[1]:
```
import os

from projectq import MainEngine
from projectq.setups import linear
from projectq.ops import H, Rx, Rz, CNOT, Measure, All

from quantuminspire.api import QuantumInspireAPI
from quantuminspire.credentials import get_authentication
from quantuminspire.projectq.backend_qx import QIBackend

QI_URL = os.getenv('API_URL', 'https://api.quantum-inspire.com/')
```

[2]:
```
authentication = get_authentication()
qi_api = QuantumInspireAPI(QI_URL, authentication)

projectq_backend = QIBackend(quantum_inspire_api=qi_api)
```

Execute algorithm on QX simulator

We create an algorithm to entangle qubit 0 and qubit 4.

[3]:
```
engine = MainEngine(backend=projectq_backend)  # create default compiler (simulator back-
↪end)

qubits = engine.allocate_qureg(5)
q1 = qubits[0]
q2 = qubits[-1]

H | q1  # apply a Hadamard gate
CNOT | (q1, q2)
All(Measure) | qubits  # measure the qubits

engine.flush()  # flush all gates (and execute measurements)

print("Measured {}".format(','.join([str(int(q)) for q in qubits])))
print('Probabilities: %s' % (projectq_backend.get_probabilities(qubits),))
print(projectq_backend.cqasm())
```

```
Measured 1,0,0,0,1
Probabilities: {'00000': 0.4921875, '10001': 0.5078125}
version 1.0
```

(continues on next page)

(continued from previous page)

```
# cQASM generated by Quantum Inspire <class 'quantuminspire.projectq.backend_qx.QIBackend
↪'> class
qubits 5

h q[0]
cnot q[0], q[4]
```

The result is as expected: about half of the results is split between 0 and 1 on qubit 0 and 4. The QASM generated by the backend is fairly simple.

Simulate a spin-qubit array

On a spin-qubit array we have limited connectivity and also a limited set of gates available. With ProjectQ we can handle these cases by adding specific compiler engines. Our engine lists is generated by the `projectq.setups.linear` module.

[4]:
```
projectq_backend = QIBackend(quantum_inspire_api=qi_api)
engine_list = linear.get_engine_list(num_qubits=5, one_qubit_gates=(Rx, Rz), two_qubit_
↪gates=(CNOT,))
engine = MainEngine(backend=projectq_backend, engine_list=engine_list)  # create default␣
↪compiler (simulator back-end)

qubits = engine.allocate_qureg(5)
q1 = qubits[0]
q2 = qubits[-1]

H | q1  # apply a Hadamard gate
CNOT | (q1, q2)
All(Measure) | qubits  # measure the qubits

engine.flush()  # flush all gates (and execute measurements)

print("Measured {}".format(','.join([str(int(q)) for q in qubits])))
print('Probabilities: %s' % (projectq_backend.get_probabilities(qubits),))
print(projectq_backend.cqasm())
```

```
Measured 1,0,0,0,1
Probabilities: {'00000': 0.5048828125, '10001': 0.4951171875}
version 1.0
# cQASM generated by Quantum Inspire <class 'quantuminspire.projectq.backend_qx.QIBackend
↪'> class
qubits 5

rx q[0],1.5707963268
rz q[0],1.5707963268
rx q[0],7.85398163397
cnot q[0], q[1]
```

The result is the same, but if we look at the QASM generated there is quite a difference. The H gate was replaced by some single qubit operations. Also the qubits 0 and 4 have been mapped to neighboring qubits.

[5]:
```
current_mapping = engine.mapper.current_mapping
for l, p in current_mapping.items():
    print('mapping logical qubit %d to physical qubit %d' % (l, p))
```

```
mapping logical qubit 0 to physical qubit 0
mapping logical qubit 4 to physical qubit 1
mapping logical qubit 1 to physical qubit 2
mapping logical qubit 2 to physical qubit 3
mapping logical qubit 3 to physical qubit 4
```

[ ]:
```
```

Grover Search Algorithm

This notebook is an adapted version from https://github.com/QISKit/qiskit-tutorial. We show to perform the Grover Search algorithm both on a local simulator and on the Quantum Inspire backend.

For more information about how to use the IBM Q Experience (QX), consult the tutorials, or check out the community.

Contributors Pieter Eendebak, Giacomo Nannicini and Rudy Raymond (based on this article)

Introduction

Grover search is one of the most popular algorithms used for searching a solution among many possible candidates using Quantum Computers. If there are N possible solutions among which there is exactly one solution (that can be verified by some function evaluation), then Grover search can be used to find the solution with $O(\sqrt{N})$ function evaluations. This is in contrast to classical computers that require $\Omega(N)$ function evaluations: the Grover search is a quantum algorithm that provably can be used search the correct solutions quadratically faster than its classical counterparts.

Here, we are going to illustrate the use of Grover search to find a particular value in a binary number. The key elements of Grovers algorithm are: 1. Initialization to a uniform superposition 2. The oracle function 3. Reflections (amplitude amplification)

[1]:
```
import numpy as np
import os

from qiskit.tools.visualization import plot_histogram
from qiskit import execute, QuantumCircuit, QuantumRegister, ClassicalRegister
from qiskit import BasicAer
from IPython.display import display, Math

from quantuminspire.credentials import get_authentication
from quantuminspire.qiskit import QI

QI_URL = os.getenv('API_URL', 'https://api.quantum-inspire.com/')
```

The oracle function

We implement an oracle function (black box) that acts as -1 on a single basis state, and +1 on all other status.

[2]:
```
def format_vector(state_vector, decimal_precision=7):
    """ Format the state vector into a LaTeX formatted string.

    Args:
        state_vector (list or array): The state vector with complex
                                      values e.g. [-1, 2j+1].

    Returns:
        str: The LaTeX format.
    """
    result = []
    epsilon = 1/pow(10, decimal_precision)
    bit_length = (len(state_vector) - 1).bit_length()
    for index, complex_value in enumerate(state_vector):
        has_imag_part = np.round(complex_value.imag, decimal_precision) != 0.0
        value = complex_value if has_imag_part else complex_value.real
        value_round = np.round(value, decimal_precision)
        if np.abs(value_round) < epsilon:
            continue

        binary_state = '{0:0{1}b}'.format(index, bit_length)
        result.append(r'{0:+2g}\left\lvert {1}\right\rangle '.format(value_round, binary_
↪state))
    return ''.join(result)
```

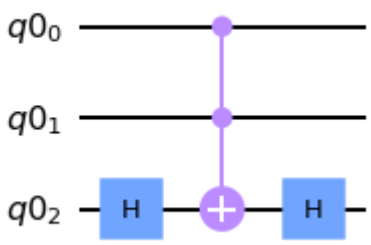
[3]:
```
def run_circuit(q_circuit, q_register, number_of_qubits=None, backend_name='statevector_
↪simulator'):
    """ Run a circuit on all base state vectors and show the output.

    Args:
        q_circuit (QuantumCircuit):
        q_register (QuantumRegister)
        number_of_qubits (int or None): The number of qubits.
        backend (str): ...
    """
    if not isinstance(number_of_qubits, int):
        number_of_qubits = q_register.size

    if q_register.size != number_of_qubits:
        warnings.warn('incorrect register size?')

    latex_text = r'\mathrm{running\ circuit\ on\ set\ of\ basis\ states:}'
    display(Math(latex_text))

    base_states = 2 ** number_of_qubits
    backend = BasicAer.get_backend(backend_name)
    for base_state in range(base_states):
        pre_circuit = QuantumCircuit(q_register)
        state = base_state
```

(continues on next page)

(continued from previous page)

```
        for kk in range(number_of_qubits):
            if state % 2 == 1:
                pre_circuit.x(q[kk])
            state = state // 2

        input_state = r'\left\lvert{0:0{1}b}\right\rangle'.format(base_state, number_of_
↪qubits)
        circuit_total = pre_circuit.compose(q_circuit)
        job = execute(circuit_total, backend=backend)
        output_state = job.result().get_statevector(circuit_total)

        latex_text = input_state + r'\mathrm{transforms\ to}: ' + format_vector(output_
↪state)
        display(Math(latex_text))
```

[4]:
```
n=3
N=2**n
q = QuantumRegister(n)
qc = QuantumCircuit(q)

if n==1:
    def black_box(qc, q):
        qc.z(q)
elif n==2:
    def black_box(qc, q):
        for i in range(n):
            qc.s(q[i])
        qc.h(q[1])
        qc.cx(q[0], q[1])
        qc.h(q[1])
        for i in range(n):
            qc.s(q[i])
else:
    def black_box(qc, q):
        qc.h(q[2])
        qc.ccx(q[0], q[1], q[2])
        qc.h(q[2])
black_box(qc,q)
cplot=qc.draw(output='mpl')
display(cplot)

print('black box circuit:')
run_circuit(qc, q)
```




```
black box circuit:
```

running circuit on set of basis states :

$|000\rangle$ transforms to : $+1\,|000\rangle$

$|001\rangle$ transforms to : $+1\,|001\rangle$

$|010\rangle$ transforms to : $+1\,|010\rangle$

$|011\rangle$ transforms to : $+1\,|011\rangle$

$|100\rangle$ transforms to : $+1\,|100\rangle$

$|101\rangle$ transforms to : $+1\,|101\rangle$

$|110\rangle$ transforms to : $+1\,|110\rangle$

$|111\rangle$ transforms to : $-1\,|111\rangle$

Inversion about the average

Another important procedure in Grover search is to have an operation that perfom the *inversion-about-the-average* step, namely, it performs the following transformation:

$$\sum_{j=0}^{2^n-1} \alpha_j |j\rangle \rightarrow \sum_{j=0}^{2^n-1} \left(2 \left(\sum_{k=0}^{k=2^n-1} \frac{\alpha_k}{2^n} \right) - \alpha_j \right) |j\rangle$$

The above transformation can be used to amplify the probability amplitude α_s when s is the solution and α_s is negative (and small), while α_j for $j \neq s$ is positive. Roughly speaking, the value of α_s increases by twice the average of the amplitudes, while others are reduced. The inversion-about-the-average can be realized with the sequence of unitary matrices as below:

$$H^{\otimes n} \left(2|0\rangle\langle 0| - I\right) H^{\otimes n}$$

The first and last H are just Hadamard gates applied to each qubit. The operation in the middle requires us to design a sub-circuit that flips the probability amplitude of the component of the quantum state corresponding to the all-zero binary string. The sub-circuit can be realized by the following function, which is a multi-qubit controlled-Z which flips the probability amplitude of the component of the quantum state corresponding to the all-one binary string. Applying X gates to all qubits before and after the function realizes the sub-circuit.

[5]:
```
def n_controlled_Z(circuit, controls, target):
    """Implement a Z gate with multiple controls"""
    if (len(controls) > 2):
        raise ValueError('The controlled Z with more than 2 ' +
                         'controls is not implemented')
```

(continues on next page)

(continued from previous page)

```
    elif (len(controls) == 1):
        circuit.h(target)
        circuit.cx(controls[0], target)
        circuit.h(target)
    elif (len(controls) == 2):
        circuit.h(target)
        circuit.ccx(controls[0], controls[1], target)
        circuit.h(target)
```

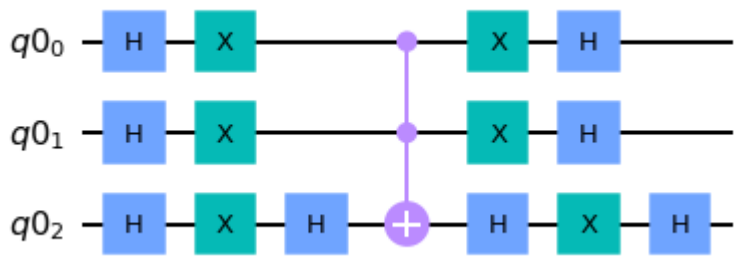
Finally, the inversion-about-the-average circuit can be realized by the following function:

[6]:
```
def inversion_about_average(circuit, f_in, n):
    """Apply inversion about the average step of Grover's algorithm."""
    # Hadamards everywhere
    if n==1:
        circuit.x(f_in[0])
        return
    for j in range(n):
        circuit.h(f_in[j])
    # D matrix: flips the sign of the state |000> only
    for j in range(n):
        circuit.x(f_in[j])
    n_controlled_Z(circuit, [f_in[j] for j in range(n-1)], f_in[n-1])
    for j in range(n):
        circuit.x(f_in[j])
    # Hadamards everywhere again
    for j in range(n):
        circuit.h(f_in[j])
```

We show the circuit that performs inversion about the average on n qubits. We also show the effect of the circuit on the basis states.

[7]:
```
qInvAvg = QuantumCircuit(q)
inversion_about_average(qInvAvg, q, n)
qInvAvg.draw(output='mpl')
```

[7]:



[8]:
```
print('inversion average circuit:')
qc = QuantumCircuit(q)
inversion_about_average(qc, q, n)
run_circuit(qc, q, n)
```

`inversion average circuit:`
running circuit on set of basis states :
$\|000\rangle$ transforms to : $+0.75\,\|000\rangle - 0.25\,\|001\rangle - 0.25\,\|010\rangle - 0.25\,\|011\rangle - 0.25\,\|100\rangle - 0.25\,\|101\rangle - 0.25\,\|110\rangle - 0.25\,\|111\rangle$
$\|001\rangle$ transforms to : $-0.25\,\|000\rangle + 0.75\,\|001\rangle - 0.25\,\|010\rangle - 0.25\,\|011\rangle - 0.25\,\|100\rangle - 0.25\,\|101\rangle - 0.25\,\|110\rangle - 0.25\,\|111\rangle$
$\|010\rangle$ transforms to : $-0.25\,\|000\rangle - 0.25\,\|001\rangle + 0.75\,\|010\rangle - 0.25\,\|011\rangle - 0.25\,\|100\rangle - 0.25\,\|101\rangle - 0.25\,\|110\rangle - 0.25\,\|111\rangle$
$\|011\rangle$ transforms to : $-0.25\,\|000\rangle - 0.25\,\|001\rangle - 0.25\,\|010\rangle + 0.75\,\|011\rangle - 0.25\,\|100\rangle - 0.25\,\|101\rangle - 0.25\,\|110\rangle - 0.25\,\|111\rangle$
$\|100\rangle$ transforms to : $-0.25\,\|000\rangle - 0.25\,\|001\rangle - 0.25\,\|010\rangle - 0.25\,\|011\rangle + 0.75\,\|100\rangle - 0.25\,\|101\rangle - 0.25\,\|110\rangle - 0.25\,\|111\rangle$
$\|101\rangle$ transforms to : $-0.25\,\|000\rangle - 0.25\,\|001\rangle - 0.25\,\|010\rangle - 0.25\,\|011\rangle - 0.25\,\|100\rangle + 0.75\,\|101\rangle - 0.25\,\|110\rangle - 0.25\,\|111\rangle$
$\|110\rangle$ transforms to : $-0.25\,\|000\rangle - 0.25\,\|001\rangle - 0.25\,\|010\rangle - 0.25\,\|011\rangle - 0.25\,\|100\rangle - 0.25\,\|101\rangle + 0.75\,\|110\rangle - 0.25\,\|111\rangle$
$\|111\rangle$ transforms to : $-0.25\,\|000\rangle - 0.25\,\|001\rangle - 0.25\,\|010\rangle - 0.25\,\|011\rangle - 0.25\,\|100\rangle - 0.25\,\|101\rangle - 0.25\,\|110\rangle + 0.75\,\|111\rangle$

Grover Search: putting all together

The complete steps of Grover search is as follow.

1. Create the superposition of all possible solutions as the initial state (with working qubits initialized to zero)

$$\sum_{j=0}^{2^n-1} \frac{1}{2^n}|j\rangle|0\rangle$$

2. Repeat for T times:
   - Apply the blackbox function
   - Apply the inversion-about-the-average function
3. Measure to obtain the solution

Before we go to the code to perform the Grover search we make some remarks on the number of repetitions T that we have to perform (for details see Grover algorithm, Wikipedia).

Each Grover step rotates the 'winner solution' by a fixed angle. This means that after a certain number of steps we arrive at the optimal approximation (e.g. the amplitude of the winner solution is maximal). If we then apply more iterations, the quality of our result will go *down*. For a database of size $N = 2^n$ the optimal number of iterations is

$$r = \pi\sqrt{N}/4$$

[9]:
```
theta = 2*np.arcsin(1/np.sqrt(N))
r=np.pi*np.sqrt(N)/4
```

(continues on next page)

(continued from previous page)

```
display(Math(r'\textrm{Rotation of the winner: } \theta = %.2f \mathrm{\ [deg]}' % (np.
↪rad2deg(theta))) )
print('Optimal number of Grover iterations for n=%d: %.1f' % (n,r) )
T=int(r)
```

Rotation of the winner: $\theta = 41.41$ [deg]

```
Optimal number of Grover iterations for n=3: 2.2
```

The probablity of the winner state after T iterations is $\sin((T + 1/2)\theta)^2$

[10]:
```
for i in range(int(r+2)):
    p=np.sin((i+1/2)*theta)**2
    print('%d iterations: p %.2f' % (i, p))
```

```
0 iterations: p 0.12
1 iterations: p 0.78
2 iterations: p 0.95
3 iterations: p 0.33
```

Finally we define the complete circuit for Grovers algorithm, excute it and show the results.

[11]:
```
"""Grover search implemented in QISKit.

This module contains the code necessary to run Grover search on 3
qubits, both with a simulator and with a real quantum computing
device. This code is the companion for the paper
"An introduction to quantum computing, without the physics",
Giacomo Nannicini, https://arxiv.org/abs/1708.03684.

"""
def input_state(circuit, f_in, n):
    """(n+1)-qubit input state for Grover search."""
    for j in range(n):
        circuit.h(f_in[j])

q = QuantumRegister(n)
ans = ClassicalRegister(n)
qc = QuantumCircuit(q, ans)

input_state(qc, q, n)

backend=BasicAer.get_backend('statevector_simulator')
job = execute(qc, backend=backend, shots=10)
result = job.result()
state_vector = result.get_statevector(qc)
m=display( Math('\mathrm{state\ after\ initialization:\ }' +format_vector(state_vector)))

# apply T rounds of oracle and inversion about the average
print('number of iterations T=%d'% T)
for t in range(T):
    for i in range(n):
        qc.barrier(q[i]) # for better visualization
    qc.i(q[0])
```

(continues on next page)

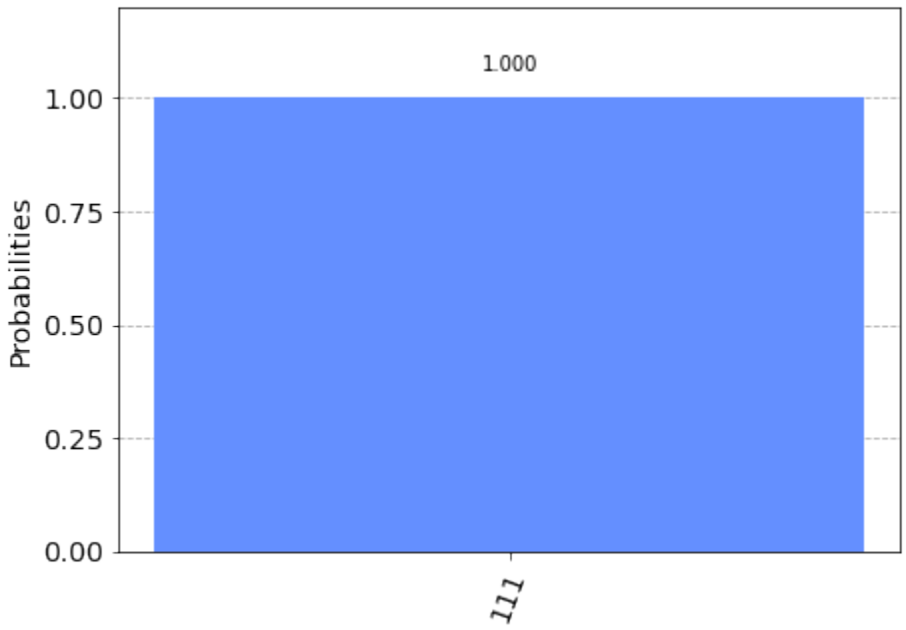
(continued from previous page)

```
    # Apply T full iterations
    black_box(qc, q)
    for i in range(n):
        qc.barrier(q[i])
    qc.i(q[0])
    inversion_about_average(qc, q, n)

# Measure the output register in the computational basis
for j in range(n):
    qc.measure(q[j], ans[j])

# Execute circuit
backend=BasicAer.get_backend('qasm_simulator')
job  = execute(qc, backend=backend, shots=10)
result = job.result()

# Get counts and plot histogram
counts = result.get_counts()
plot_histogram(counts)
```

state after initialization : $+0.353553\,|000\rangle + 0.353553\,|001\rangle + 0.353553\,|010\rangle + 0.353553\,|011\rangle + 0.353553\,|100\rangle + 0.353553\,|101\rangle + 0.353553\,|110\rangle + 0.353553\,|111\rangle$

```
number of iterations T=2
```

[11]:



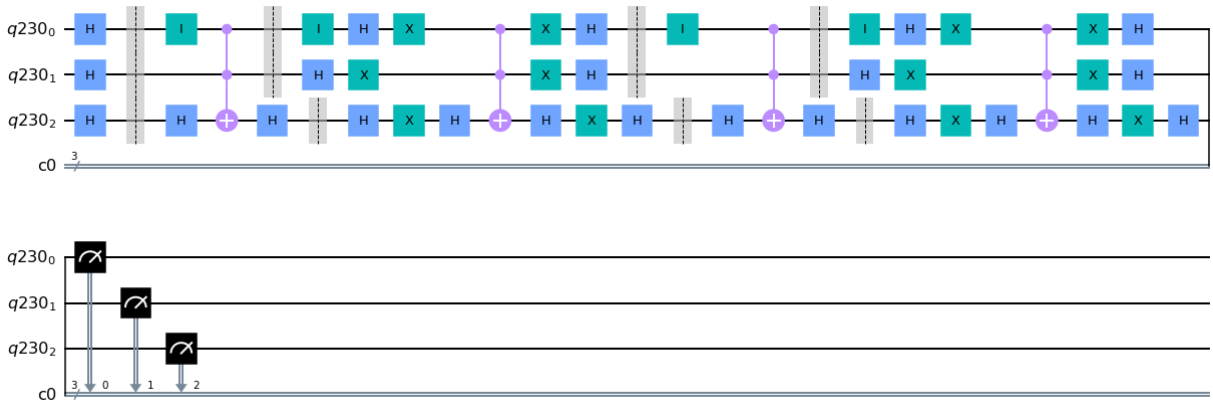
As expected, the state that is indicated by the oracle function has the highest probability of begin measured.

We show the full circuit that was generated by the code.

[12]:

```
qc.draw(output='mpl')
```

[12]:



Run the circuit on the Quantum Inspire simulator

First we make a connection to the Quantum Inspire website.

[13]:
```
authentication = get_authentication()
QI.set_authentication(authentication, QI_URL)
```

We can list backends and perform other functions with the `QuantumInspireProvider`.

[14]:
```
QI.backends()
```

[14]:
```
[<QuantumInspireBackend('QX-34-L') from QI()>,
 <QuantumInspireBackend('Spin-2') from QI()>,
 <QuantumInspireBackend('QX single-node simulator') from QI()>,
 <QuantumInspireBackend('Starmon-5') from QI()>]
```

We create a QisKit backend for the Quantum Inspire interface and execute the circuit generated above.

[15]:
```
qi_backend = QI.get_backend('QX single-node simulator')
j=execute(qc, backend=backend, shots=512)
```

We can wait for the results and then print them

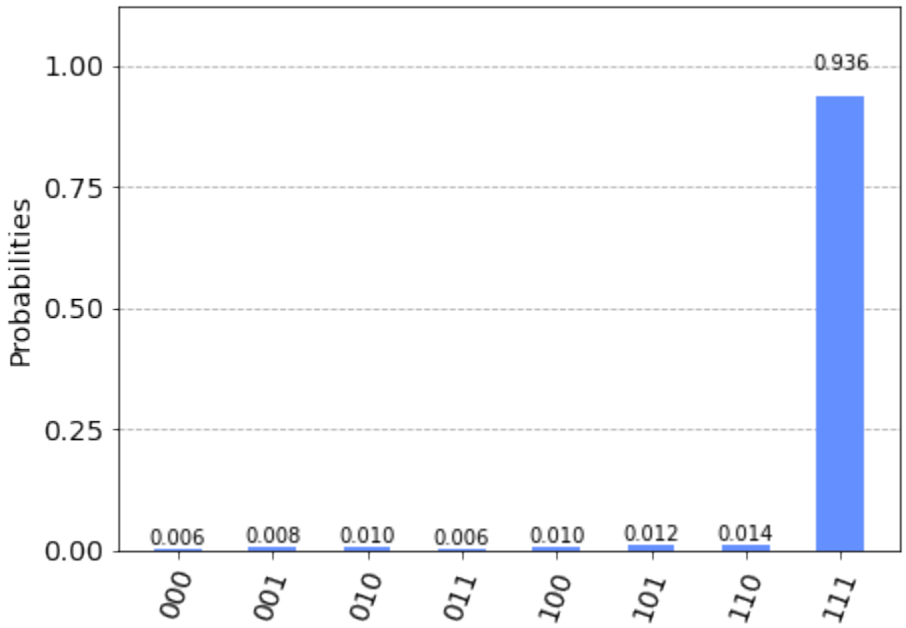
[16]:
```
result = j.result()
print('Generated histogram:')
print(result.get_counts())
```

```
Generated histogram:
{'111': 479, '010': 5, '100': 5, '110': 7, '001': 4, '101': 6, '000': 3, '011': 3}
```

Visualization can be done with the normal Python plotting routines, or with the QisKit SDK.

[17]:
```
plot_histogram(result.get_counts(qc))
```

[17]:



A screenshot from the execution result on the Quantum Inspire website.

References

[1] "A fast quantum mechanical algorithm for database search", L. K. Grover, Proceedings of the 28th Annual ACM Symposium on the Theory of Computing (STOC 1996)

[2] "Tight bounds on quantum searching", Boyer et al., Fortsch.Phys.46:493-506,1998

[3] "Quantum Inspire"

[ ]:

Quantum Inspire performance test

We compare performance of the simulator with the circuit from

"Overview and Comparison of Gate Level Quantum Software Platforms", https://arxiv.org/abs/1807.02500

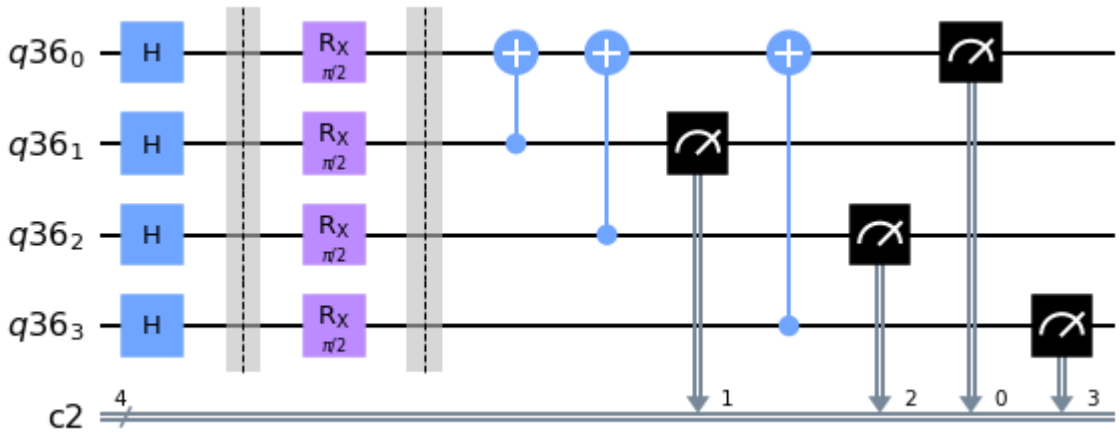
Define the circuit

[9]:
```
import time
import os
import numpy as np

from qiskit import QuantumCircuit, ClassicalRegister, QuantumRegister, execute
from qiskit.tools.visualization import plot_histogram

from quantuminspire.credentials import get_authentication
from quantuminspire.qiskit import QI

QI_URL = os.getenv('API_URL', 'https://api.quantum-inspire.com/')
```

We define the circuit based on the number of qubits and the depth (e.g. the number of iterations of the unit building block).

[11]:
```
def pcircuit(nqubits, depth = 10):
    """ Circuit to test performance of quantum computer """
    q = QuantumRegister(nqubits)
    ans = ClassicalRegister(nqubits)
    qc = QuantumCircuit(q, ans)

    for level in range(depth):
        for qidx in range(nqubits):
            qc.h( q[qidx] )
        qc.barrier()
        for qidx in range(nqubits):
            qc.rx(np.pi/2, q[qidx])
        qc.barrier()

        for qidx in range(nqubits):
            if qidx!=0:
                qc.cx(q[qidx], q[0])
    for qidx in range(nqubits):
        qc.measure(q[qidx], ans[qidx])
    return q, qc

q,qc = pcircuit(4, 1)
qc.draw(output='mpl')
```

[11]:



Run the cirquit on the Quantum Inspire simulator

First we make a connection to the Quantum Inspire website.

[12]:
```
authentication = get_authentication()
QI.set_authentication(authentication, QI_URL)
```

We create a QisKit backend for the Quantum Inspire interface and execute the circuit generated above.

[13]:
```
qi_backend = QI.get_backend('QX single-node simulator')
job = execute(qc, qi_backend)
```

We can wait for the results and then print them

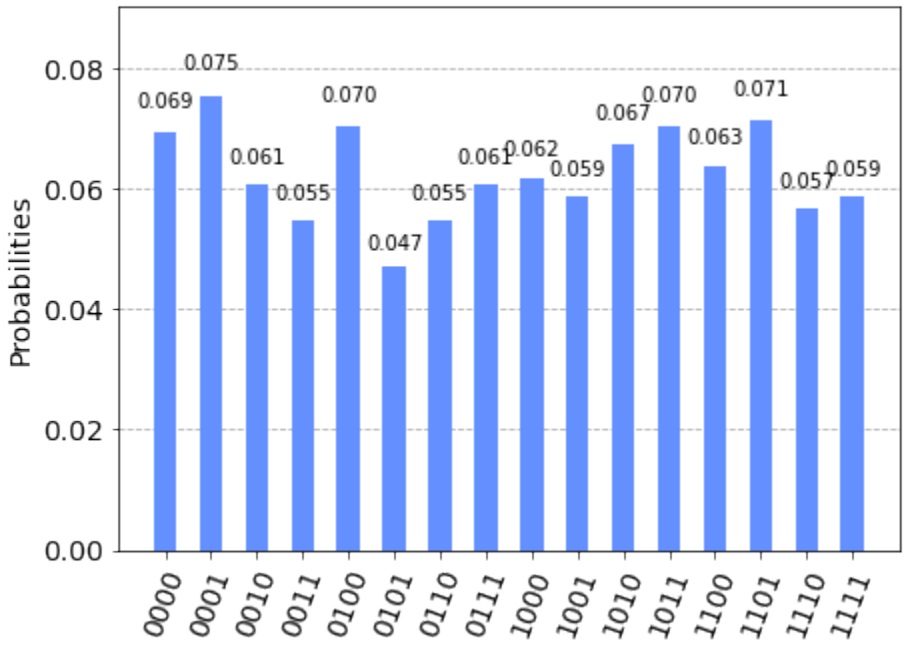
[14]:
```
result = job.result()
print('Generated histogram:')
print(result.get_counts())
```

```
Generated histogram:
{'0000': 71, '0001': 77, '0010': 62, '0011': 56, '0100': 72, '0101': 48, '0110': 56,
↪'0111': 62, '1000': 63, '1001': 60, '1010': 69, '1011': 72, '1100': 65, '1101': 73,
↪'1110': 58, '1111': 60}
```

Visualization can be done with the normal Python plotting routines, or with the QisKit SDK.

[15]:
```
plot_histogram(result.get_counts(qc))
```

[15]:



To compare we will run the circuit with 20 qubits and depth 20. This takes:

- QisKit: 3.7 seconds
- ProjectQ: 2.0 seconds

Our simulator runs for multiple shots (unless full state projection is used). More details will follow later.

[16]:
```
q, qc = pcircuit(10, 10)
start_time = time.time()

job = execute(qc, qi_backend, shots=8)
job.result()
interval = time.time() - start_time

print('time needed: %.1f [s]' % (interval,))
```

```
time needed: 3.9 [s]
```

[ ]:

1.2.2 Classifier examples

A Quantum distance-based classifier (part 1)

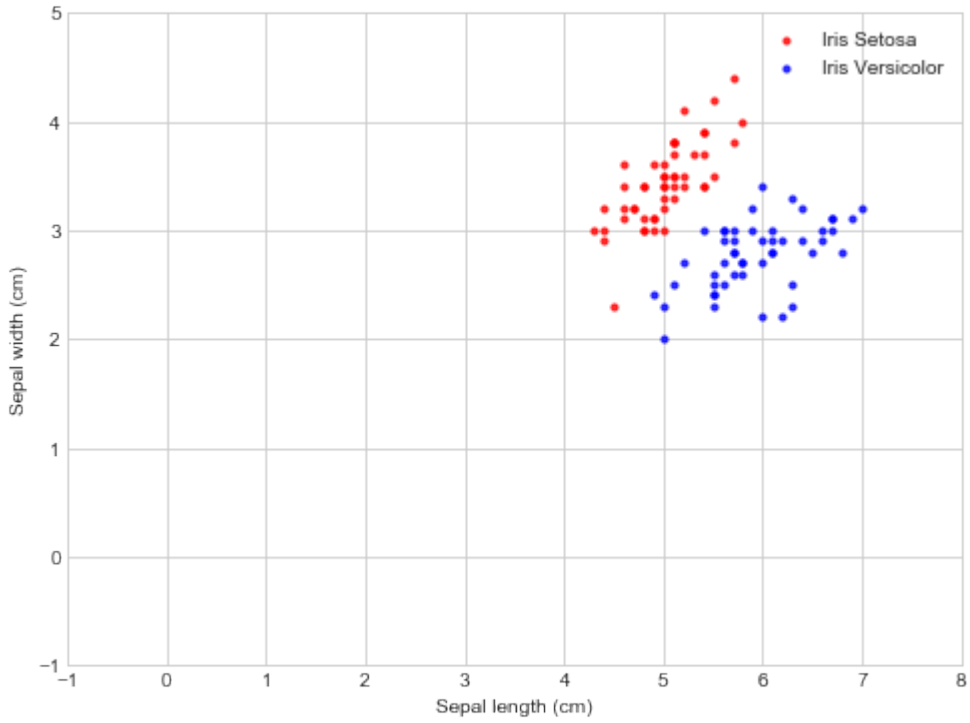
Robert Wezeman, TNO

Table of Contents

[1]:
```
# Import external python file
import numpy as np
from data_plotter import get_bin, DataPlotter  # for easier plotting
DataPlotter = DataPlotter()
```

Introduction

Consider the following scatter plot of the first two flowers in the famous Iris flower data set



Notice that just two features, the sepal width and the sepal length, divide the two different Iris species into different regions in the plot. This gives rise to the question: given only the sepal length and sepal width of a flower can we classify the flower by their correct species? This type of problem, also known as statistical classification, is a common problem in machine learning. In general, a classifier is constructed by letting it learn a function which gives the desired output based on a sufficient amount of data. This is called supervised learning, as the desired output (the labels of

the data points) are known. After learning, the classifier can classify an unlabeled data point based on the learned function. The quality of a classifier improves if it has a larger training dataset it can learn on. The true power of this quantum classifier becomes clear when using extremely large data sets. In this notebook we will describe how to build a distance-based classifier on the Quantum Inspire using amplitude encoding. It turns out that, once the system is initialized in the desired state, regardless of the size of training data, the actual algorithm consists of only 3 actions, one Hadamard gate and two measurements. This has huge implications for the scalability of this problem for large data sets. Using only 4 qubits we show how to encode two data points, both of a different class, to predict the label for a third data point. In this notebook we will demonstrate how to use the Quantum Inspire SDK using QASM-code, we will also provide the code to obtain the same results for the ProjectQ framework.

Back to Table of Contents

Problem

We define the following binary classification problem: Given the data set

$$\mathcal{D} = \Big\{(\mathbf{x}_1, y_1), \dots (\mathbf{x}_M, y_M)\Big\},$$

consisting of M data points $x_i \in \mathbb{R}^n$ and corresponding labels $y_i \in \{-1, 1\}$, give a prediction for the label $\tilde{y}$ corresponding to an unlabeled data point $\tilde{\mathbf{x}}$. The classifier we shall implement with our quantum circuit is a distance-based classifier and is given by

$$\tilde{y} = \text{sgn}\left(\sum_{m=0}^{M-1} y_m \left[1 - \frac{1}{4M}\left|\tilde{\mathbf{x}} - \mathbf{x}_m\right|^2\right]\right). \qquad (1) \qquad (1.1)$$

This is a typical M-nearest-neighbor model, where each data point is given a weight related to the distance measure. To implement this classifier on a quantum computer, we need a way to encode the information of the training data set in a quantum state. We do this by first encoding the training data in the amplitudes of a quantum system, and then manipulate the amplitudes of then the amplitudes will be manipulated by quantum gates such that we obtain a result representing the above classifier. Encoding input features in the amplitude of a quantum system is known as amplitude encoding.

Back to Contents

Amplitude encoding

Suppose we want to encode a classical vector $\mathbf{x} \in \mathbb{R}^{\mathbf{N}}$ by some amplitudes of a quantum system. We assume $N = 2^n$ and that $\mathbf{x}$ is normalised to unit length, meaning $\mathbf{x}^{\mathbf{T}}\mathbf{x} = 1$. We can encode $\mathbf{x}$ in the amplitudes of a n-qubit system in the following way

$$\mathbf{x} = \begin{pmatrix} x^1 \\ \vdots \\ x^N \end{pmatrix} \Longleftrightarrow \psi_{\mathbf{x}} = \sum_{i=0}^{N-1} x^i i, \qquad (1.2)$$

where i is the i^{th} entry of the computational basis $\{0 \dots 0, \dots, 1 \dots 1\}$. By applying an efficient quantum algorithm (resources growing polynomially in the number of qubits n), one can manipulate the 2^n amplitudes super efficiently, that is $\mathcal{O}(\log N)$. This follows as manipulating all amplitudes requires an operation on each of the $n = \mathcal{O}(\log N)$ qubits. For algorithms to be truly super-efficient, the phase where the data is encoded must also be at most polynomial in the number of qubits. The idea of quantum memory, sometimes referred as quantum RAM (QRAM), is a particular interesting one. Suppose we first run some quantum algorithm, for example in quantum chemistry, with as output some resulting quantum states. If these states could be fed into a quantum classifier, the encoding phase is not needed anymore. Finding efficient data encoding systems is still a topic of active research. We will restrict ourselves here to the implementation of the algorithm, more details can be found in the references.

The algorithm requires the n-qubit quantum system to be in the following state

$$\mathcal{D} = \frac{1}{\sqrt{2M}} \sum_{m=0}^{M-1} m\Big(0\psi_{\tilde{\mathbf{x}}} + 1\psi_{\mathbf{x_m}}\Big)y_m. \qquad (2) \tag{1.3}$$

Here m is the m^{th} state of the computational basis used to keep track of the m^{th} training input. The second register is a single ancillary qubit entangled with the third register. The excited state of the ancillary qubit is entangled with the m^{th} training state ψ_{x_m}, while the ground state is entangled with the new input state $\psi_{\tilde{x}}$. The last register encodes the label of the m^{th} training data point by

$$\begin{aligned} y_m = -1 \Longleftrightarrow & y_m = 0, \\ y_m = 1 \Longleftrightarrow & y_m = 1. \end{aligned} \tag{1.4}$$

Once in this state the algorithm only consists of the following three operations:

1. Apply a Hadamard gate on the second register to obtain

$$\frac{1}{2\sqrt{M}} \sum_{m=0}^{M-1} m\Big(0\psi_{\tilde{\mathbf{x}}+\mathbf{x_m}} + 1\psi_{\tilde{\mathbf{x}}-\mathbf{x_m}}\Big)y_m,$$

where $\psi_{\tilde{\mathbf{x}}\pm\mathbf{x_m}} = \psi_{\tilde{\mathbf{x}}} \pm \psi_{\mathbf{x_m}}$.

2. Measure the second qubit. We restart the algorithm if we measure a 1 and only continue if we are in the 0 branch. We continue the algorithm with a probability $p_{acc} = \frac{1}{4M}\sum_M \tilde{\mathbf{x}} + \mathbf{x}_m{}^2$, for standardised random data this is usually around 0.5. The resulting state is given by

$$\frac{1}{2\sqrt{Mp_{acc}}} \sum_{m=0}^{M-1} \sum_{i=0}^{N-1} m0\left(\tilde{x}^i + x_m^i\right) iy_m. \tag{1.5}$$

3. Measure the last qubit y_m. The probability that we measure outcome zero is given by

$$p(q_4 = 0) = \frac{1}{4Mp_{acc}} \sum_{m|y_m=0} \tilde{\mathbf{x}} + \mathbf{x_m}{}^2. \tag{1.6}$$

In the special case where the amount of training data for both labels is equal, this last measurement relates to the classifier as described in previous section by

$$\tilde{y} = \begin{cases} -1 & : p(q_4 = 0) > p(q_4 = 1) \\ +1 & : p(q_4 = 0) < p(q_4 = 1) \end{cases} \tag{1.7}$$

By setting $\tilde{y}$ to be the most likely outcome of many measurement shots, we obtain the desired distance-based classifier.

Back to Table of Contents

Data preprocessing

In the previous section we saw that for amplitude encoding we need a data set which is normalised. Luckily, it is always possible to bring data to this desired form with some data transformations. Firstly, we standardise the data to have zero mean and unit variance, then we normalise the data to have unit length. Both these steps are common methods in machine learning. Effectively, we only have to consider the angle between different data features.

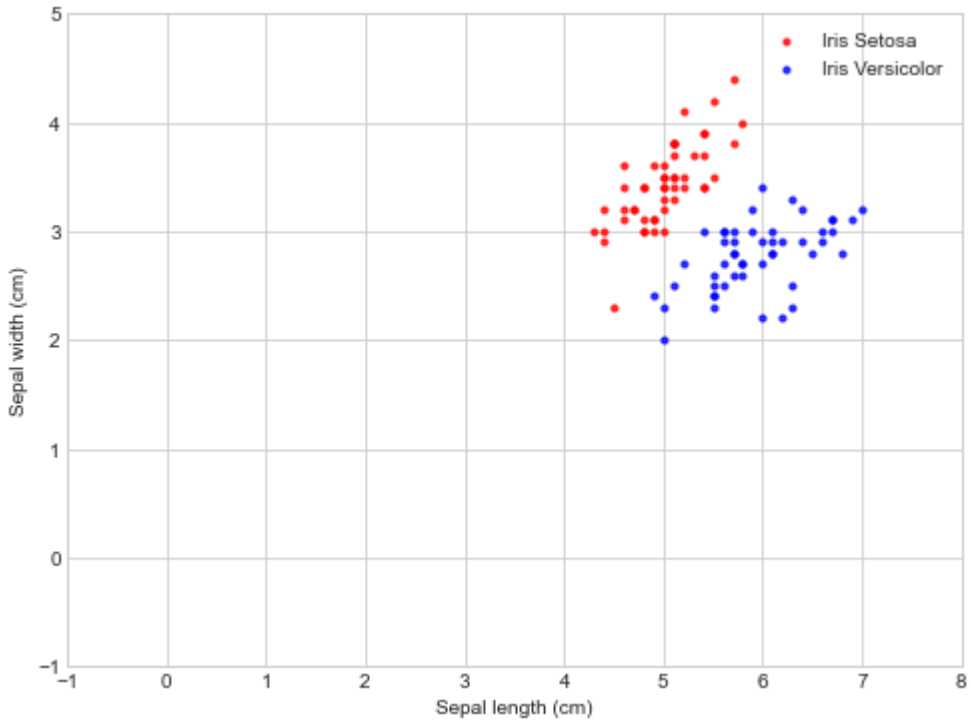
To illustrate this procedure we apply it to the first two features of the famous Iris data set:

[2]:
```
# Plot the data
from sklearn.datasets import load_iris

iris = load_iris()
features = iris.data.T
```

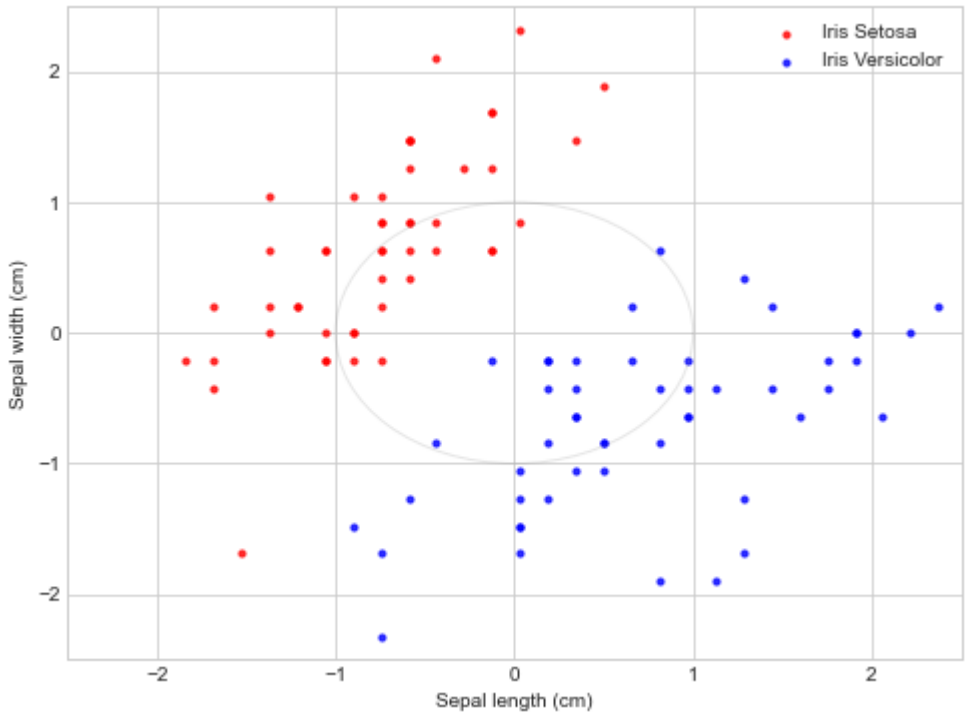
(continues on next page)

(continued from previous page)

```
data = [el[0:101] for el in features][0:2]  # Select only the first two features of the␣
↪dataset

half_len_data = len(data[0]) // 2
iris_setosa = [el[0:half_len_data] for el in data[0:2]]
iris_versicolor = [el[half_len_data:-1] for el in data[0:2]]

DataPlotter.plot_original_data(iris_setosa, iris_versicolor);  # Function to plot the␣
↪data
```



[3]:
```
# Rescale the data
from sklearn import preprocessing  # Module contains method to rescale data to have zero␣
↪mean and unit variance

# Rescale whole data-set to have zero mean and unit variance
features_scaled = [preprocessing.scale(el) for el in data[0:2]]
iris_setosa_scaled = [el[0:half_len_data] for el in features_scaled]
iris_versicolor_scaled = [el[half_len_data:-1] for el in features_scaled]

DataPlotter.plot_standardised_data(iris_setosa_scaled, iris_versicolor_scaled);  #␣
↪Function to plot the data
```



[4]:
```
# Normalise the data
def normalise_data(arr1, arr2):
    """Normalise data to unit length
        input: two array same length
        output: normalised arrays
    """
    for idx in range(len(arr1)):
        norm = (arr1[idx]**2 + arr2[idx]**2)**(1 / 2)
        arr1[idx] = arr1[idx] / norm
        arr2[idx] = arr2[idx] / norm
    return [arr1, arr2]


iris_setosa_normalised = normalise_data(iris_setosa_scaled[0], iris_setosa_scaled[1])
iris_versicolor_normalised = normalise_data(iris_versicolor_scaled[0], iris_versicolor_
↪scaled[1])
# Function to plot the data
DataPlotter.plot_normalised_data(iris_setosa_normalised, iris_versicolor_normalised);
```

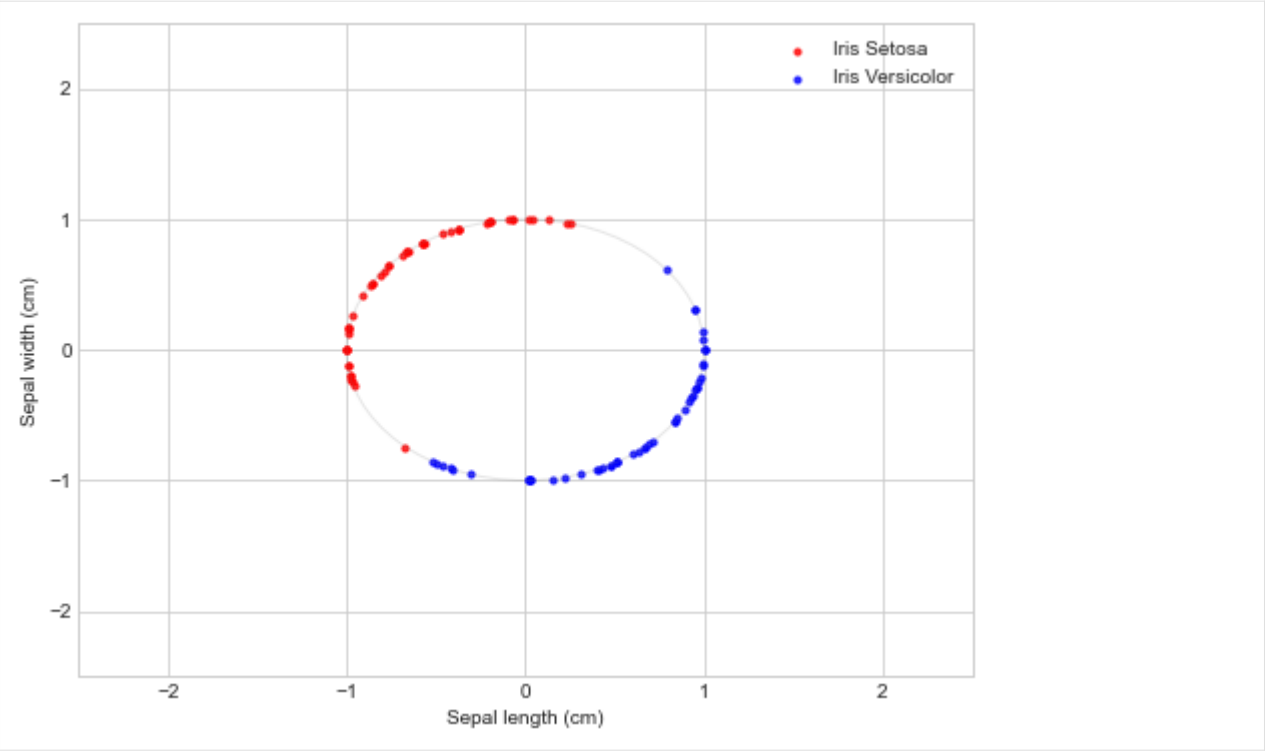


Table of Contents

Quantum algorithm

Now we can start with our quantum algorithm on the Quantum Inspire. We describe how to build the algorithm for the simplest case with only two data points, each with two features, that is $M = N = 2$. For this algorithm we need 4 qubits:

- One qubit for the index register m
- One ancillary qubit
- One qubit to store the information of the two features of the data points
- One qubit to store the information of the classes of the data points

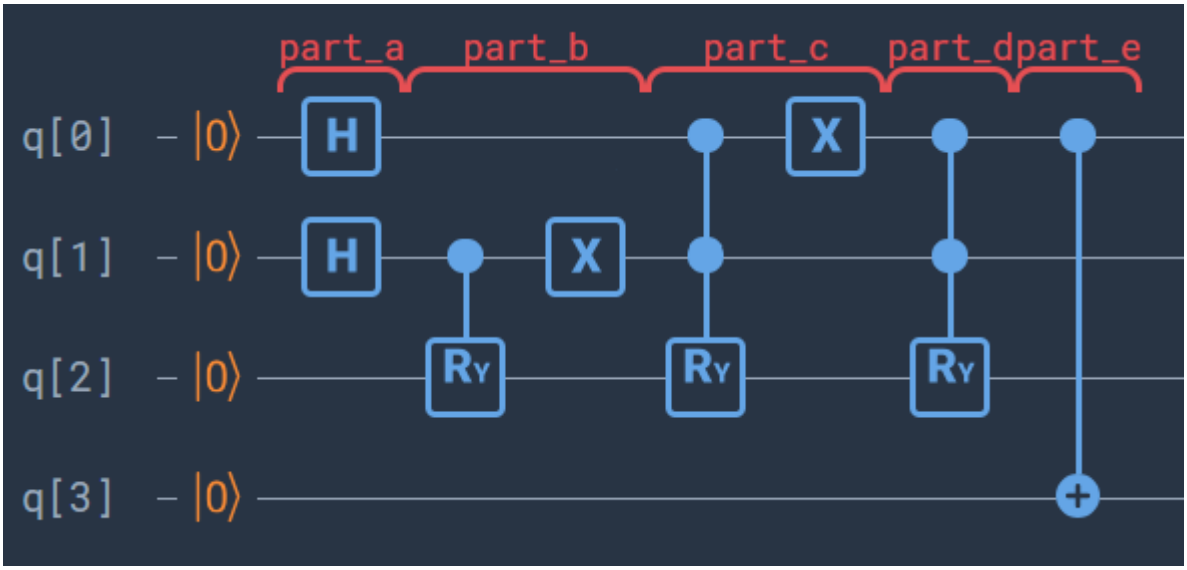
From the data set described in previous section we pick the following data set $\mathcal{D} = \{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2)\}$ where:

- $\mathbf{x}_1 = (0.9193, 0.3937)$, $y_1 = -1$,
- $\mathbf{x}_2 = (0.1411, 0.9899)$, $y_2 = 1$.

We are interested in the label $\tilde{y}$ for the data point $\tilde{\mathbf{x}} = (0.8670, 0.4984)$. The amplitude encoding of these data points look like

$$\begin{aligned} \psi_{\tilde{\mathbf{x}}} &= 0.86700 + 0.49841, \\ \psi_{\mathbf{x_1}} &= 0.91930 + 0.39371, \\ \psi_{\mathbf{x_2}} &= 0.14110 + 0.98991. \end{aligned} \tag{1.8}$$

Before we can run the actual algorithm we need to bring the system in the desired *initial state (equation 2)* which can be obtain by applying the following combination of gates starting on 0000.



- **Part A:** In this part the index register is initialized and the ancilla qubit is brought in the desired state. For this we use the plain QASM language of the Quantum Inspire. Part A consists of two Hadamard gates:

[5]:
```
def part_a():
    qasm_a = """version 1.0
qubits 4
prep_z q[0:3]
.part_a
H q[0:1] #execute Hadamard gate on qubit 0, 1
"""
    return qasm_a
```

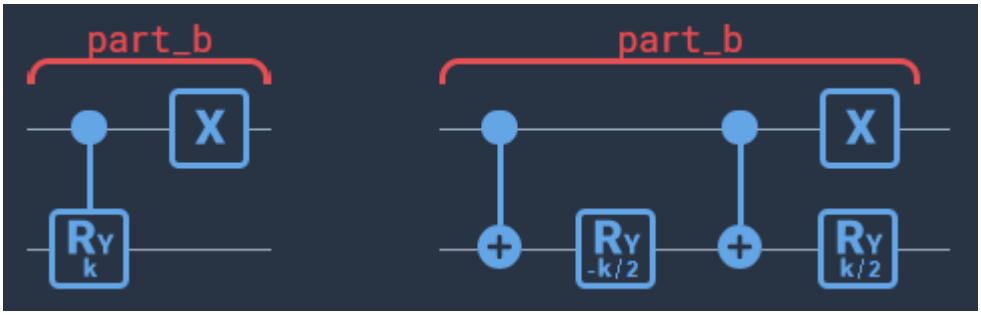
After this step the system is in the state

$$\mathcal{D}_A = \frac{1}{2}\Big(0+1\Big)\Big(0+1\Big)00$$

- **Part B:** In this part we encode the unlabeled data point $\tilde{x}$ by making use of a controlled rotation. We entangle the third qubit with the ancillary qubit. The angle θ of the rotation should be chosen such that $\tilde{x} = R_y(\theta)0$. By the definition of R_y we have

$$R_y(\theta)0 = \cos\left(\frac{\theta}{2}\right)0 + \sin\left(\frac{\theta}{2}\right)1.$$

Therefore, the angle needed to rotate to the state $\psi = a0 + b1$ is given by $\theta = 2\cos^{-1}(a) \cdot sign(b)$. Quantum Inspire does not directly support controlled-R_y gates, however we can construct it from other gates as shown in the figure below. In these pictures k stand for the angle used in the R_y rotation.

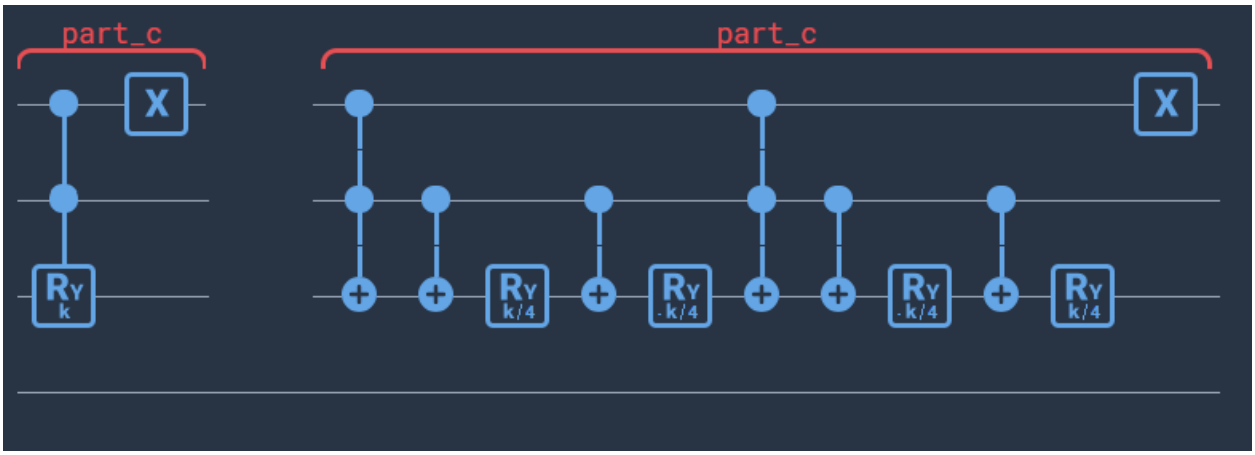


[6]:
```
def part_b(angle):
    half_angle = angle / 2
    qasm_b = """.part_b # encode test value x^tilde
CNOT q[1], q[2]
Ry q[2], -{0}
CNOT q[1], q[2]
Ry q[2], {0}
X q[1]
""".format(half_angle)
    return qasm_b
```

After this step the system is in the state

$$\mathcal{D}_B = \frac{1}{2}\Big(0+1\Big)\Big(0\tilde{x}+10\Big)0$$

- **Part C:** In this part we encode the first data point x_1. The rotation angle θ is such that $x_1 = R_y(\theta)0$. Now a double controlled-R_y rotation is needed, and similar to Part B, we construct it from other gates as shown in the figure below.



[7]:
```
def part_c(angle):
    quarter_angle = angle / 4
    qasm_c = """.part_c # encode training x^0 value
toffoli q[0],q[1],q[2]
CNOT q[0],q[2]
Ry q[2], {0}
CNOT q[0],q[2]
Ry q[2], -{0}
toffoli q[0],q[1],q[2]
CNOT q[0],q[2]
Ry q[2], -{0}
CNOT q[0],q[2]
Ry q[2], {0}
X q[0]
""".format(quarter_angle)
    return qasm_c
```

After this step the system is in the state

$$\mathcal{D}_C = \frac{1}{2}\left(0\Big(0\tilde{x}+1x_1\Big)+1\Big(0\tilde{x}+10\Big)\right)0$$

- **Part D:** This part is almost an exact copy of part C, however now with θ chosen such that $x_2 = R_y(\theta)0$.

[8]:
```
def part_d(angle):
    quarter_angle = angle / 4
    qasm_d = """.part_d # encode training x^1 value
toffoli q[0],q[1],q[2]
CNOT q[0],q[2]
Ry q[2], {0}
CNOT q[0],q[2]
Ry q[2], -{0}
toffoli q[0],q[1],q[2]
CNOT q[0],q[2]
Ry q[2], -{0}
CNOT q[0],q[2]
Ry q[2], {0}
""".format(quarter_angle)
    return qasm_d
```

After this step the system is in the state

$$\mathcal{D}_D = \frac{1}{2}\left(0\Big(0\tilde{x} + 1x_1\Big) + 1\Big(0\tilde{x} + 1x_2\Big)\right)0$$

- **Part E:** The last step is to label the last qubit with the correct class, this can be done using a simple CNOT gate between the first and last qubit to obtain the desired initial state

$$\mathcal{D}_E = \frac{1}{2}0\Big(0\tilde{x} + 1x_1\Big)0 + 1\Big(0\tilde{x} + 1x_2\Big)1.$$

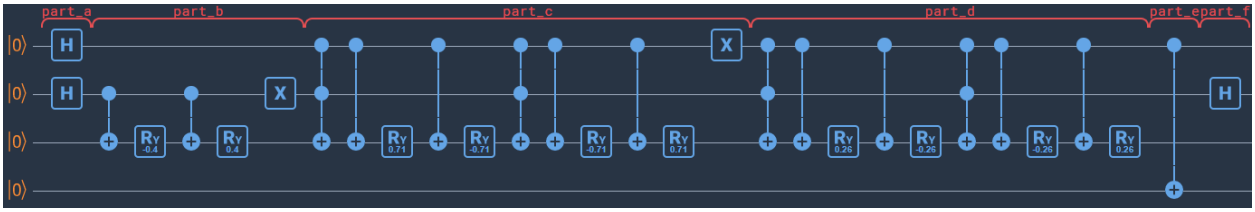
[9]:
```
def part_e():
    qasm_e = """.part_e # encode the labels
CNOT q[0], q[3]
"""
    return qasm_e
```

The actual algorithm

Once the system is in this initial state, the algorithm itself only consists of one Hadamard gate and two measurements. If the first measurement gives the result 1, we have to abort the algorithm and start over again. However, these results can also easily be filtered out in a post-proecessing step.

[10]:
```
def part_f():
    qasm_f = """
.part_f
H q[1]
"""
    return qasm_f
```

The circuit for the whole algorithm now looks like:



We can send our QASM code to the Quantum Inspire with the following data points

$$\begin{aligned} \psi_{\tilde{x}} &= 0.86700 + 0.49841, \\ \psi_{x_1} &= 0.91930 + 0.39371, \\ \psi_{x_2} &= 0.14110 + 0.98991. \end{aligned} \tag{1.9}$$

[11]:
```
import os
from quantuminspire.api import QuantumInspireAPI
from quantuminspire.credentials import get_authentication
from math import acos
from math import pi

QI_URL = os.getenv('API_URL', 'https://api.quantum-inspire.com/')

# input data points:
angle_x_tilde = 2 * acos(0.8670)
angle_x0 = 2 * acos(0.1411)
angle_x1 = 2 * acos(0.9193)


authentication = get_authentication()
qi = QuantumInspireAPI(QI_URL, authentication)

# Build final QASM
final_qasm = part_a() + part_b(angle_x_tilde) + part_c(angle_x0) + part_d(angle_x1) +␣
↪part_e() + part_f()

backend_type = qi.get_backend_type_by_name('QX single-node simulator')
result = qi.execute_qasm(final_qasm, backend_type=backend_type, number_of_shots=1, full_
↪state_projection=True)

print(result['histogram'])
```

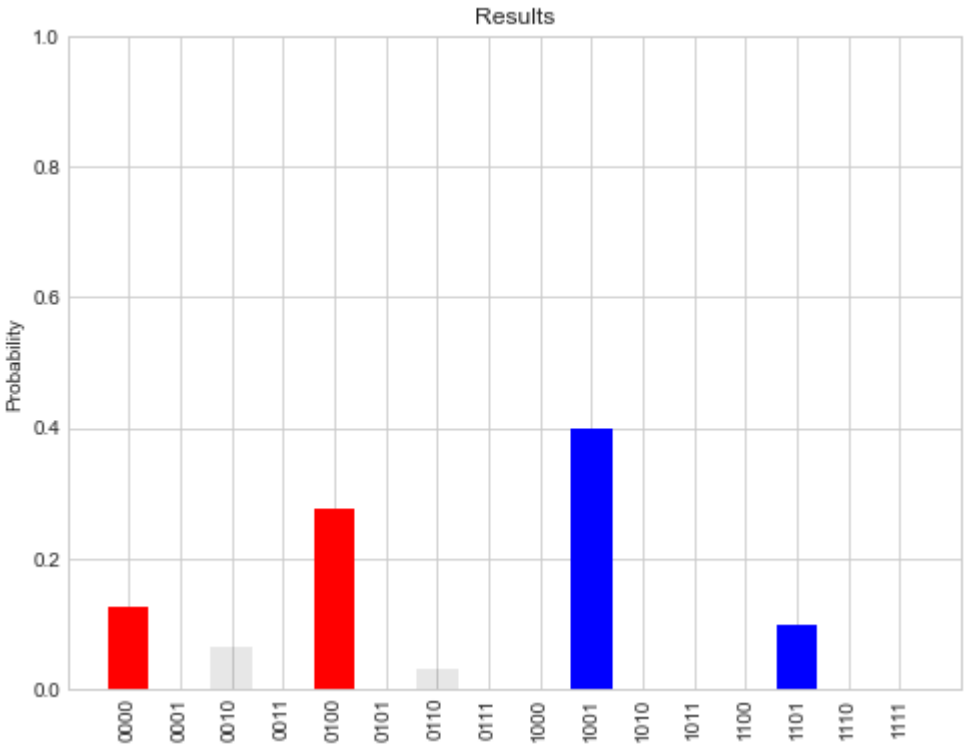
```
OrderedDict([('9', 0.3988584), ('4', 0.2768809), ('0', 0.1270332), ('13', 0.099428), ('2
↪', 0.0658663), ('6', 0.0302195), ('15', 0.0013716), ('11', 0.0003419)])
```

[12]:
```
import matplotlib.pyplot as plt


def bar_plot(result_data):
    res = [get_bin(el, 4) for el in range(16)]
    prob = [0] * 16

    for key, value in result_data['histogram'].items():
```

(continues on next page)

(continued from previous page)

```
        prob[int(key)] = value

    # Set color=light grey when 2nd qubit = 1
    # Set color=blue when 2nd qubit = 0, and last qubit = 1
    # Set color=red when 2nd qubit = 0, and last qubit = 0
    color_list = [
        'red', 'red', (0.1, 0.1, 0.1, 0.1), (0.1, 0.1, 0.1, 0.1),
        'red', 'red', (0.1, 0.1, 0.1, 0.1), (0.1, 0.1, 0.1, 0.1),
        'blue', 'blue', (0.1, 0.1, 0.1, 0.1), (0.1, 0.1, 0.1, 0.1),
        'blue', 'blue', (0.1, 0.1, 0.1, 0.1), (0.1, 0.1, 0.1, 0.1)
                ]
    plt.bar(res, prob, color=color_list)
    plt.ylabel('Probability')
    plt.title('Results')
    plt.ylim(0, 1)
    plt.xticks(rotation='vertical')
    plt.show()
    return prob


prob = bar_plot(result)
```



We only consider the events where the second qubit equals 0, that is, we only consider the events in the set

$$\{0000, 0001, 0100, 0101, 1000, 1001, 1100, 1101\}$$

The label $\tilde{y}$ is now given by

$$\tilde{y} = \begin{cases} -1 & : \#\{0000, 0001, 0100, 0101\} > \#\{1000, 1001, 1100, 1101\} \\ +1 & : \#\{1000, 1001, 1100, 1101\} > \#\{0000, 0001, 0100, 0101\} \end{cases} \tag{1.10}$$

[13]:
```
def summarize_results(prob, display=1):
    sum_label0 = prob[0] + prob[1] + prob[4] + prob[5]
    sum_label1 = prob[8] + prob[9] + prob[12] + prob[13]

    def y_tilde():
        if sum_label0 > sum_label1:
            return 0, ">"
        elif sum_label0 < sum_label1:
            return 1, "<"
        else:
            return "undefined", "="
    y_tilde_res, sign = y_tilde()
    if display:
        print("The sum of the events with label 0 is: {}".format(sum_label0))
        print("The sum of the events with label 1 is: {}".format(sum_label1))
        print("The label for y_tilde is: {} because sum_label0 {} sum_label1".format(y_
↪tilde_res, sign))
    return y_tilde_res


summarize_results(prob);
```

```
The sum of the events with label 0 is: 0.4039141
The sum of the events with label 1 is: 0.4982864
The label for y_tilde is: 1 because sum_label0 < sum_label1
```

The following code will randomly pick two training data points and a random test point for the algorithm. We can compare the prediction for the label by the Quantum Inspire with the true label.

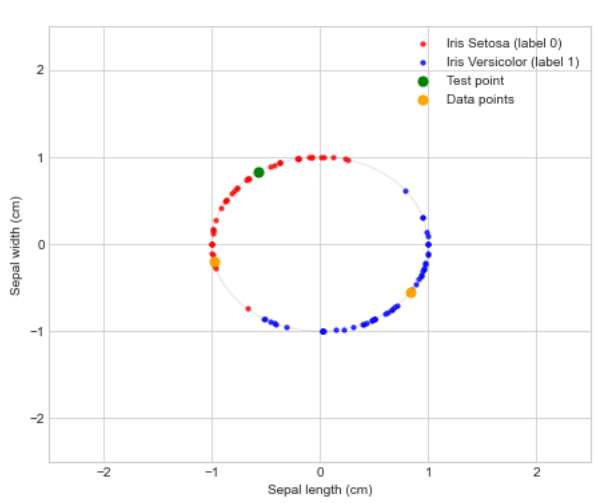
[14]:
```
from random import sample
from numpy import sign


def grab_random_data():
    one_random_index = sample(range(50), 1)
    two_random_index = sample(range(50), 2)
    random_label = sample([1,0], 1) # random label

    # iris_setosa_normalised  # Label 0
    # iris_versicolor_normalised  # Label 1
    if random_label[0]:
        # Test data has label = 1, iris_versicolor
        data_label0 = [iris_setosa_normalised[0][one_random_index[0]],
                       iris_setosa_normalised[1][one_random_index[0]]]
        data_label1 = [iris_versicolor_normalised[0][two_random_index[0]],
                       iris_versicolor_normalised[1][two_random_index[0]]]
        test_data = [iris_versicolor_normalised[0][two_random_index[1]],
                     iris_versicolor_normalised[1][two_random_index[1]]]
    else:
        # Test data has label = 0, iris_setosa
        data_label0 = [iris_setosa_normalised[0][two_random_index[0]],
                       iris_setosa_normalised[1][two_random_index[0]]]
```

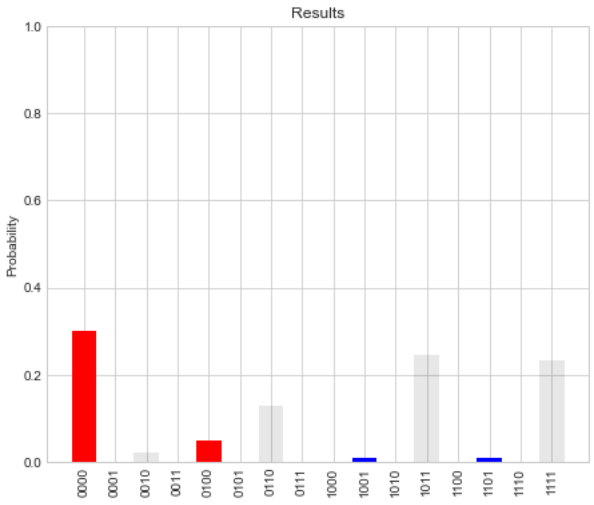
(continues on next page)

(continued from previous page)

```
        data_label1 = [iris_versicolor_normalised[0][one_random_index[0]],
                       iris_versicolor_normalised[1][one_random_index[0]]]
        test_data = [iris_setosa_normalised[0][two_random_index[1]],
                     iris_setosa_normalised[1][two_random_index[1]]]
    return data_label0, data_label1, test_data, random_label


data_label0, data_label1, test_data, random_label = grab_random_data()

print("Data point {} from label 0".format(data_label0))
print("Data point {} from label 1".format(data_label1))
print("Test point {} from label {} ".format(test_data, random_label[0]))


def run_random_data(data_label0, data_label1, test_data):
    angle_x_tilde = 2 * acos(test_data[0]) * sign(test_data[1]) % (4 * pi)
    angle_x0 = 2 * acos(data_label0[0]) * sign(data_label0[1]) % (4 * pi)
    angle_x1 = 2 * acos(data_label1[0])* sign(data_label1[1]) % (4 * pi)

    # Build final QASM
    final_qasm = part_a() + part_b(angle_x_tilde) + part_c(angle_x0) + part_d(angle_x1)␣
↪+ part_e() + part_f()
    result_random_data = qi.execute_qasm(final_qasm, backend_type=backend_type, number_
↪of_shots=1, full_state_projection=True)
    return result_random_data


result_random_data = run_random_data(data_label0, data_label1, test_data);

# Plot data points:
plt.rcParams['figure.figsize'] = [16, 6]  # Plot size
plt.subplot(1, 2, 1)
DataPlotter.plot_normalised_data(iris_setosa_normalised, iris_versicolor_normalised);
plt.scatter(test_data[0], test_data[1], s=50, c='green');  # Scatter plot data class ?
plt.scatter(data_label0[0], data_label0[1], s=50, c='orange');  # Scatter plot data␣
↪class 0
plt.scatter(data_label1[0], data_label1[1], s=50, c='orange');  # Scatter plot data␣
↪class 1
plt.legend(["Iris Setosa (label 0)", "Iris Versicolor (label 1)", "Test point", "Data␣
↪points"])
plt.subplot(1, 2, 2)
prob_random_points = bar_plot(result_random_data);
summarize_results(prob_random_points);
```

```
Data point [-0.9804333429271337, -0.19685136544287737] from label 0
Data point [0.83663763675258, -0.5477567569360124] from label 1
Test point [-0.5687658377271257, 0.8224994965558099] from label 0
```






```
The sum of the events with label 0 is: 0.3489316
The sum of the events with label 1 is: 0.0184048
The label for y_tilde is: 0 because sum_label0 > sum_label1
```

To get a better idea how well this quantum classifier works we can compare the predicted label to the true label of the test datapoint. Errors in the prediction can have two causes. The quantum classifier does not give the right classifier prediction or the quantum classifier gives the right classifier prediction which for the selected data gives the wrong label. in general, the first type of errors can be reduced by increasing the number of times we run the algorithm. In our case, as we work with the simulator and our gates are deterministic (no conditional gates), we do not have to deal with this first error if we use the true probability distribution. This can be done by using only a single shot without measurements.

[15]:
```
quantum_score = 0
error_prediction = 0
classifier_is_quantum_prediction = 0
classifier_score = 0
no_label = 0


def true_classifier(data_label0, data_label1, test_data):
    if np.linalg.norm(np.array(data_label1) - np.array(test_data)) < np.linalg.norm(np.
↪array(data_label0) -
                                                                                    np.
↪array(test_data)):
        return 1
    else:
        return 0


for idx in range(100):
    data_label0, data_label1, test_data, random_label = grab_random_data()
    result_random_data = run_random_data(data_label0, data_label1, test_data)
    classifier = true_classifier(data_label0, data_label1, test_data)

    sum_label0 = 0
    sum_label1 = 0
```

(continues on next page)

(continued from previous page)

```
    for key, value in result_random_data['histogram'].items():
        if int(key) in [0, 1, 4, 5]:
            sum_label0 += value
        if int(key) in [8, 9, 12, 13]:
            sum_label1 += value
    if sum_label0 > sum_label1:
        quantum_prediction = 0
    elif sum_label1 > sum_label0:
        quantum_prediction = 1
    else:
        no_label += 1
        continue


    if quantum_prediction == classifier:
        classifier_is_quantum_prediction += 1

    if random_label[0] == classifier:
        classifier_score += 1

    if quantum_prediction == random_label[0]:
        quantum_score += 1
    else:
        error_prediction += 1

print("In this sample of 100 data points:")
print("the classifier predicted the true label correct", classifier_score, "% of the␣
↪times")
print("the quantum classifier predicted the true label correct", quantum_score, "% of␣
↪the times")
print("the quantum classifier predicted the classifier label correct",
      classifier_is_quantum_prediction, "% of the times")
print("Could not assign a label ", no_label, "times")
```

```
In this sample of 100 data points:
the classifier predicted the true label correct 93 % of the times
the quantum classifier predicted the true label correct 93 % of the times
the quantum classifier predicted the classifier label correct 100 % of the times
Could not assign a label  0 times
```

Conclusion and further work

How well the quantum classifier performs, hugely depends on the chosen data points. In case the test data point is significantly closer to one of the two training data points the classifier will result in a one-sided prediction. The other case, where the test data point has a similar distance to both training points, the classifier struggles to give an one-sided prediction. Repeating the algorithm on the same data points, might sometimes give different measurement outcomes. This type of error can be improved by running the algorithm using more shots. In the examples above we only used the true probability distribution (as if we had used an infinite number of shots). By running the algorithm instead with 512 or 1024 shots this erroneous behavior can be observed. In case of an infinite number of shots, we see that the quantum classifier gives the same prediction as classically expected.

The results of this toy example already shows the potential of a quantum computer in machine learning. Because the

actual algorithm consists of only three operations, independent of the size of the data set, it can become extremely useful for tasks such as pattern recognition on large data sets. The next step is to extend this toy model to contain more data features and a larger training data set to improve the prediction. As not all data sets are best classified by a distance-based classifier, implementations of other types of classifiers might also be interesting. For more information on this particular classifier see the reference ref.

Back to Table of Contents

References

- Book: Schuld and Petruccione, Supervised learning with Quantum computers, 2018
- Article: Schuld, Fingerhuth and Petruccione, Implementing a distance-based classifier with a quantum interference circuit, 2017

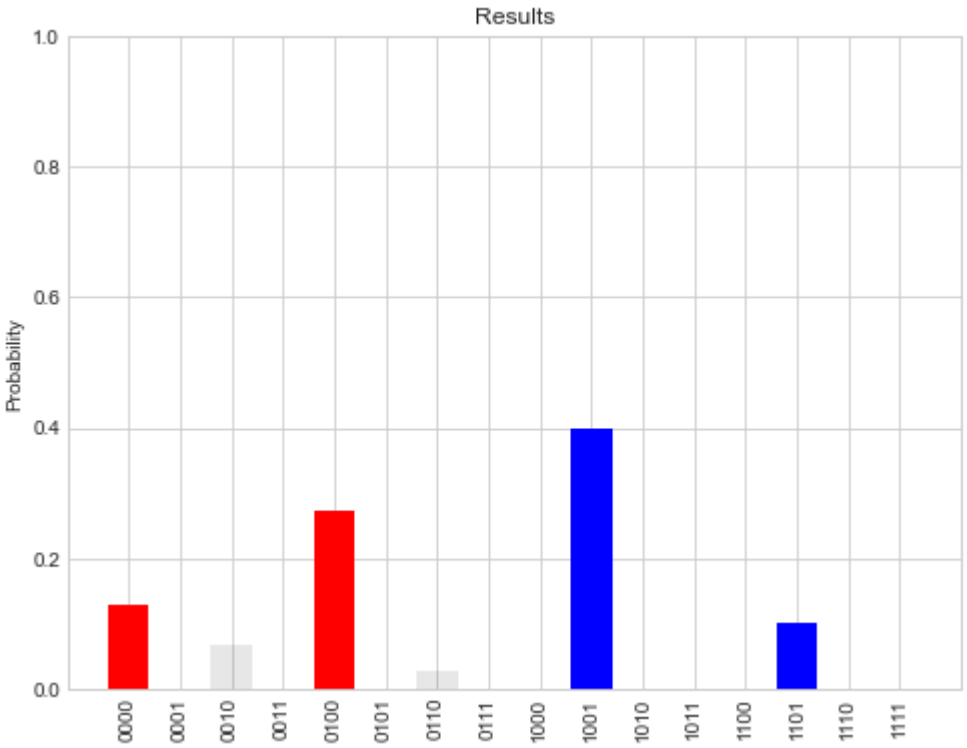
The same algorithm for the projectQ framework

[16]:
```
from math import acos
import os

from quantuminspire.api import QuantumInspireAPI
from quantuminspire.projectq.backend_qx import QIBackend

from projectq import MainEngine
from projectq.backends import ResourceCounter
from projectq.ops import CNOT, CZ, H, Toffoli, X, Ry, C
from projectq.setups import restrictedgateset

from quantuminspire.credentials import get_authentication

QI_URL = os.getenv('API_URL', 'https://api.quantum-inspire.com/')


# Remote Quantum Inspire backend #
authentication = get_authentication()
qi_api = QuantumInspireAPI(QI_URL, authentication)

compiler_engines = restrictedgateset.get_engine_list(one_qubit_gates="any",
                                                     two_qubit_gates=(CNOT, CZ, Toffoli))
compiler_engines.extend([ResourceCounter()])

qi_backend = QIBackend(quantum_inspire_api=qi_api)
qi_engine = MainEngine(backend=qi_backend, engine_list=compiler_engines)

# angles data points:
angle_x_tilde = 2 * acos(0.8670)
angle_x0 = 2 * acos(0.1411)
angle_x1 = 2 * acos(0.9193)

qubits = qi_engine.allocate_qureg(4)

# part_a
```

(continues on next page)

(continued from previous page)

```
for qubit in qubits[0:2]:
    H | qubit

# part_b
C(Ry(angle_x_tilde), 1) | (qubits[1], qubits[2]) # Alternatively build own CRy gate as␣
↪done above
X | qubits[1]

# part_c
C(Ry(angle_x0), 2) | (qubits[0], qubits[1], qubits[2]) # Alternatively build own CCRy␣
↪gate as done above
X | qubits[0]

# part_d
C(Ry(angle_x1), 2) | (qubits[0], qubits[1], qubits[2]) # Alternatively build own CCRy␣
↪gate as done above

# part_e
CNOT | (qubits[0], qubits[3])

# part_f
H | qubits[1]

qi_engine.flush()

# Results:
temp_results = qi_backend.get_probabilities(qubits)

res = [get_bin(el, 4) for el in range(16)]
prob = [0] * 16
for key, value in temp_results.items():
    prob[int(key[::-1], 2)] = value  # Reverse as projectQ has a different qubit ordering

color_list = [
    'red', 'red', (0.1, 0.1, 0.1, 0.1), (0.1, 0.1, 0.1, 0.1),
    'red', 'red', (0.1, 0.1, 0.1, 0.1), (0.1, 0.1, 0.1, 0.1),
    'blue', 'blue', (0.1, 0.1, 0.1, 0.1), (0.1, 0.1, 0.1, 0.1),
    'blue', 'blue', (0.1, 0.1, 0.1, 0.1), (0.1, 0.1, 0.1, 0.1)
              ]
plt.bar(res, prob, color=color_list)
plt.ylabel('Probability')
plt.title('Results')
plt.ylim(0, 1)
plt.xticks(rotation='vertical')
plt.show()
print("Results:")
print(temp_results)
```




```
Results:
{'0000': 0.1298828125, '0010': 0.2734375, '1001': 0.3994140625, '1011': 0.1015625, '0100
↪': 0.068359375, '0110': 0.0263671875, '1101': 0.0009765625}
```

[ ]:

A Quantum distance-based classifier (part 2)

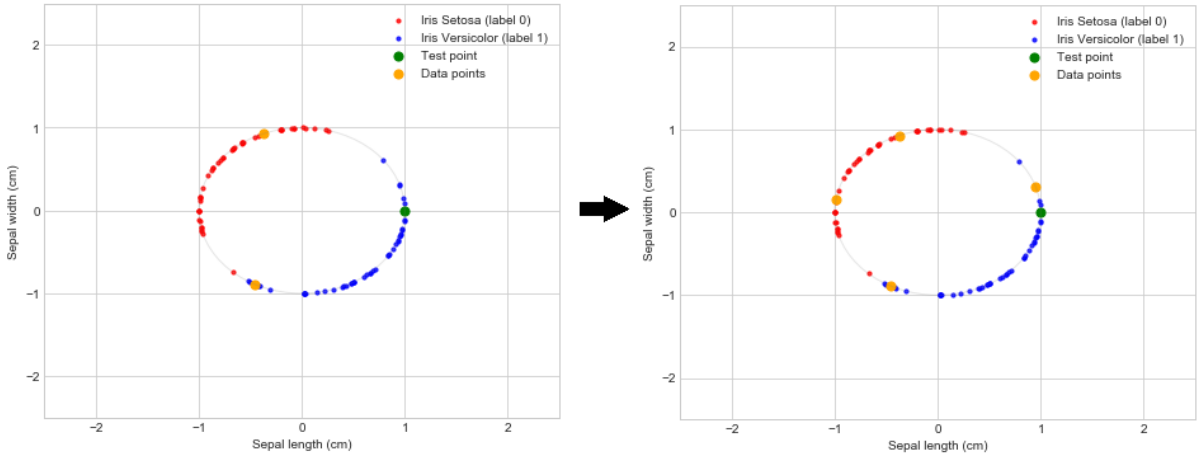
Robert Wezeman, TNO

Table of Contents

[1]:
```
## Import external python file
from data_plotter import DataPlotter  # for easier plotting
DataPlotter = DataPlotter()

# Import math functions
from math import acos
from numpy import sign
```

Introduction

In the first part of this notebook series on a distance-based classifier, we looked at how to implement a distance-based classifier on the quantum inspire using QASM code. We looked at the simplest possible case, that is: using two data points, each with two features, to assign a label (classify) to a random test point, see image. In this notebook we will extend the previous classifier, by increasing the number of data points to four. In this notebook we demonstrate how to use the projectQ framework to do this.



Back to Table of Contents

Problem

Again we have the following binary classification problem: Given the data set

$$\mathcal{D} = \Big\{(\mathbf{x}_1, y_1), \dots (\mathbf{x}_M, y_M)\Big\},$$

consisting of M data points $x_i \in \mathbb{R}^n$ and corresponding labels $y_i \in \{-1, 1\}$, give a prediction for the label $\tilde{y}$ corresponding to an unlabeled data point $\tilde{\mathbf{x}}$. The classifier we shall implement with our quantum circuit is a distance-based classifier and is given by

$$\tilde{y} = \text{sgn}\left(\sum_{m=0}^{M-1} y_m \left[1 - \frac{1}{4M}\left|\tilde{\mathbf{x}} - \mathbf{x}_m\right|^2\right]\right). \qquad (1) \qquad (1.11)$$

This is a typical M-nearest-neighbor model where each data point is given a weight related to the distance measure. To implement this classifier on a quantum computer we use amplitude encoding. For the details see the previous notebook.

Back to Contents

Theory

The algorithm requires a n-qubit quantum system to be in the following state initially

$$\mathcal{D} = \frac{1}{\sqrt{2M}} \sum_{m=0}^{M-1} m\Big(0\psi_{\tilde{\mathbf{x}}} + 1\psi_{\mathbf{x_m}}\Big)y_m. \qquad (2) \tag{1.12}$$

Here m is the m^{th} state of the computational basis used to keep track of the m^{th} training input. The second register is a single ancillary qubit entangled with the third register. The excited state of the ancillary qubit is entangled with the m^{th} training state ψ_{x_m}, while the ground state is entangled with the new input state $\psi_{\tilde{x}}$. The last register encodes the label of the m^{th} training data point by

$$\begin{aligned} y_m = -1 \Longleftrightarrow y_m = 0, \\ y_m = 1 \Longleftrightarrow y_m = 1. \end{aligned} \tag{1.13}$$

Once in this state the algorithm only consists of the following three operations:

1. Apply a Hadamard gate on the second register.
2. Measure the second register. We restart the algorithm if we measure a 1 and only continue when we are in the 0 branch.
3. Measure the last qubit y_m.

In the special case where the amount of training data for both labels is the same, this last measurement relates to the classifier as described in previous section by

$$\tilde{y} = \begin{cases} -1 & : p(q_4 = 0) > p(q_4 = 1) \\ +1 & : p(q_4 = 0) < p(q_4 = 1) \end{cases} \tag{1.14}$$

By setting $\tilde{y}$ to be the most likely outcome of many measurement shots, we obtain the desired distance-based classifier.

In the previous notebook we saw the implementation for $N = 2$ data points, each with $M = 2$ features. Now we will consider the case for two datapoints with $M = 4$ features.

Back to Table of Contents

Algorithm

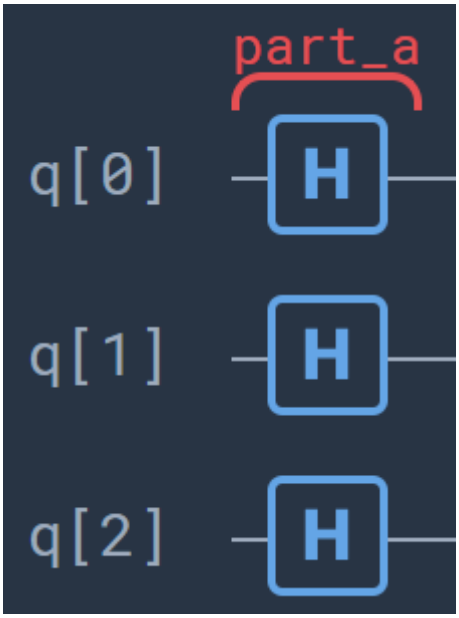
To describe the desired initial state for $M = 4$ and $N = 2$ we need 5 qubits:

- Two qubits for the index register m
- One ancillary qubit
- One qubit to store the information of the two features of the data points
- One qubit to store the information of the classes of the data points

Furthermore, these 4 require us to implement a triple controlled-R_y gate, or $CCCR_y$. ProjectQ does this automatically for us, at the cost of two extra ancillary qubits, resulting in a total of 7 qubits. The algorithm is divided in different parts:

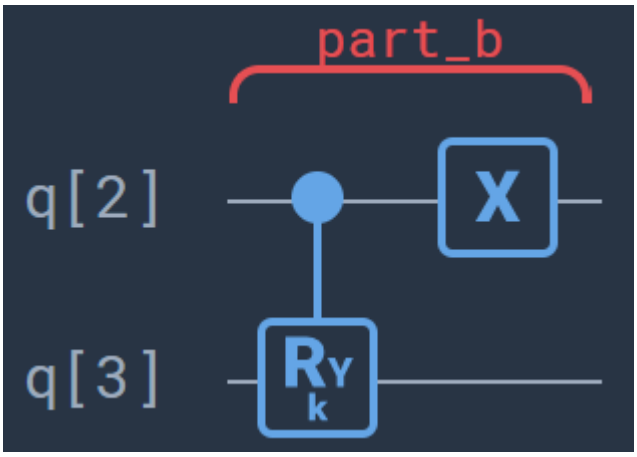
- **Part A:** In this part the index register is initialized, as is the ancillary qubit. Part A consists of three Hadamard gates. After this step the system is in the state

$$\mathcal{D}_A = \frac{1}{\sqrt{8}} \sum_{m=0}^{3} m\Big(0 + 1\Big)00$$



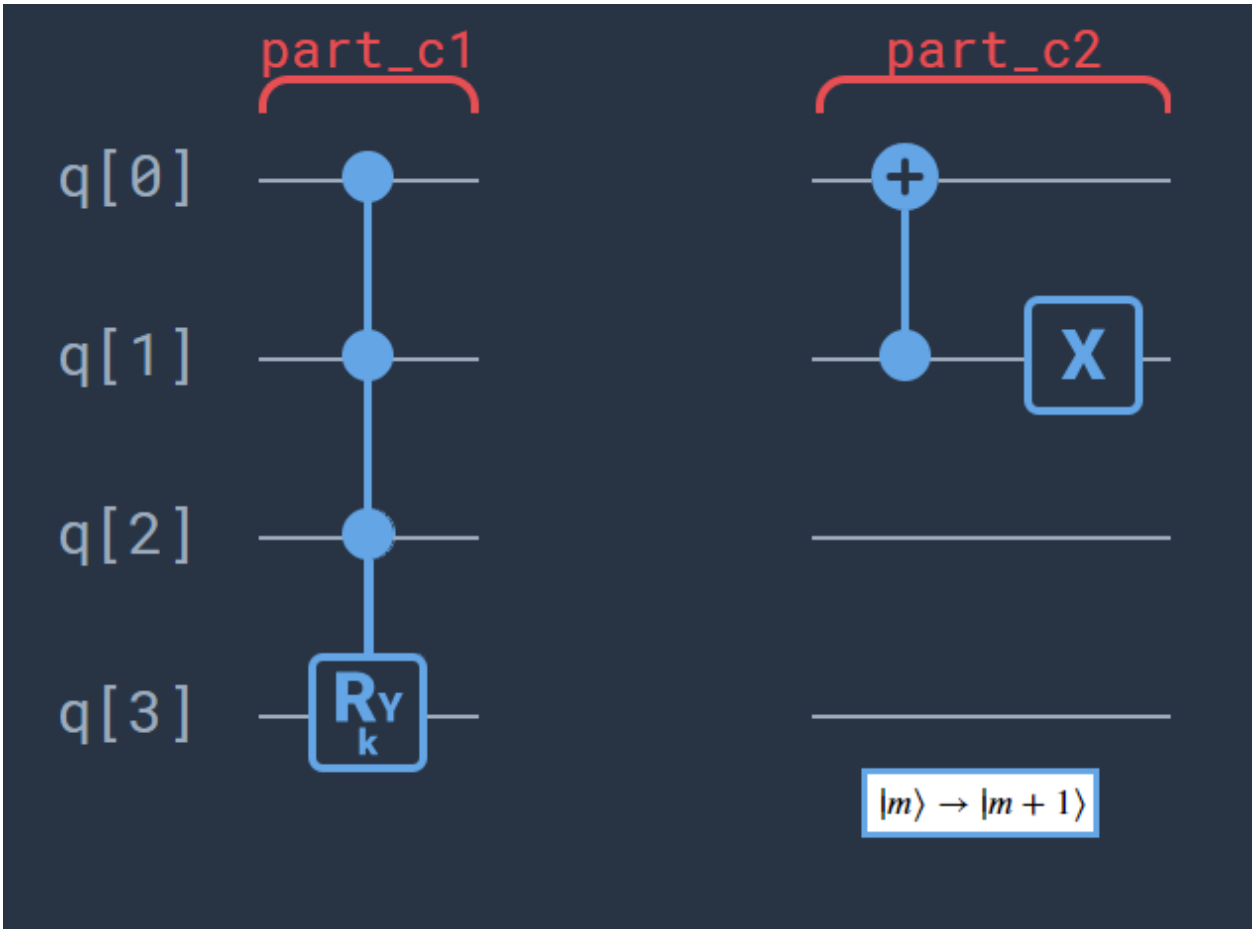
- **Part B:** In this part we encode the unlabeled data point $\tilde{\mathbf{x}}$ by means of a controlled rotation. The encoding of the data will first require some preprocessing on the data, see the previous notebook for the details. We entangle the fourth qubit with the ancillary qubit. The angle θ of the rotation should be chosen such that $\tilde{\mathbf{x}} = R_y(\theta)0$. After this step the system is in the state

$$\mathcal{D}_B = \frac{1}{\sqrt{8}} \sum_{m=0}^{3} m\Big(0\tilde{\mathbf{x}} + 10\Big)0$$



- **Part C:** In this step the encoding of the data points $\mathbf{x_m}$ is done. So far it has been almost analogous to the $M = 2$ case, however, this step is a bit more involved. First, use a $CCCR_y$-rotation so that the datapoint $\mathbf{x_m}$ is connected to $m = 11$ and entangled with the ancillary qubit 1. Next, we rotate m cyclic around such that we can reuse this $CCCR_y$-rotation for next data point, see the figure

$$C2: \qquad \ldots \rightarrow 00 \rightarrow 01 \rightarrow 10 \rightarrow 11 \rightarrow \ldots \tag{1.15}$$

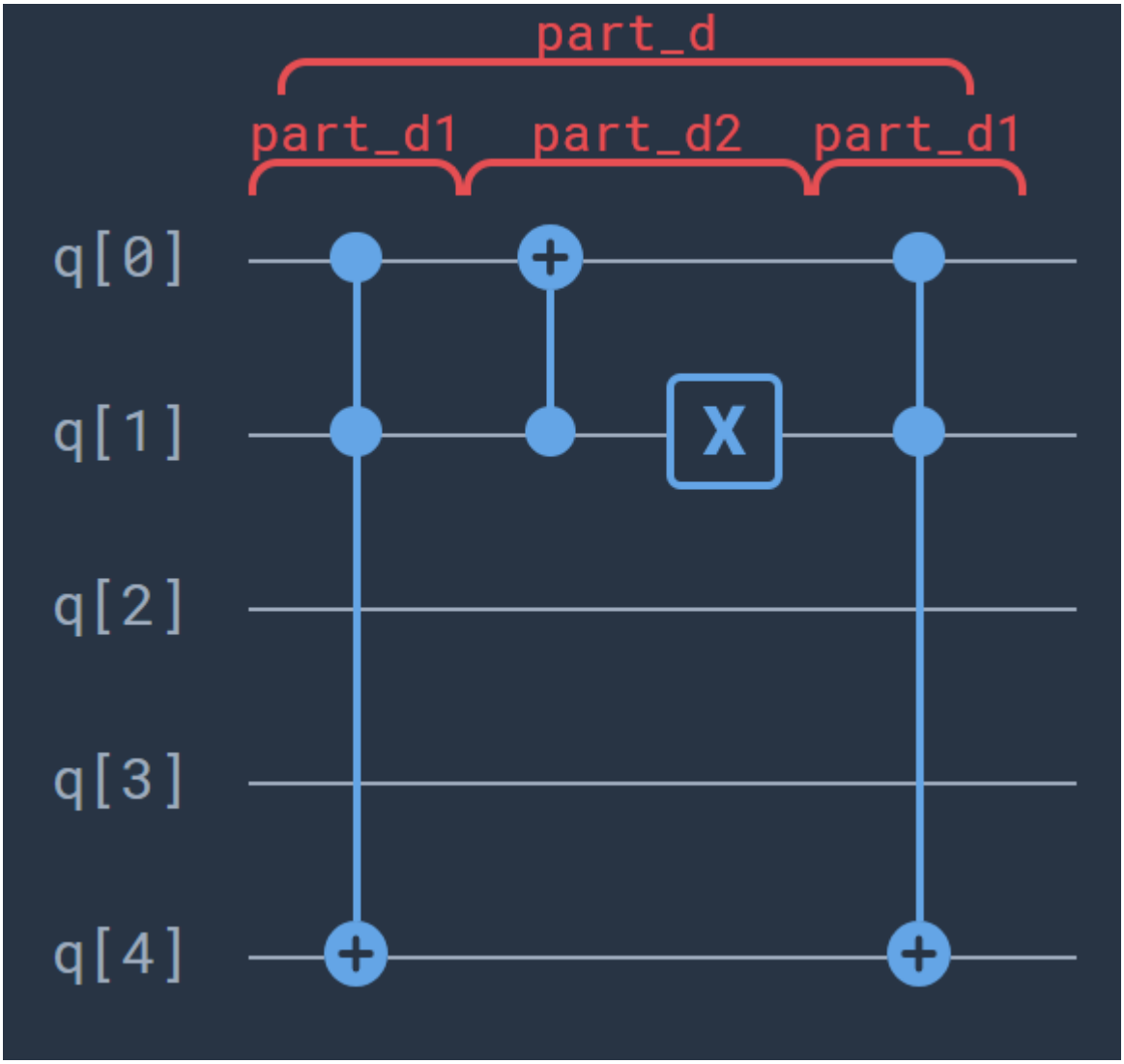


After doing step $C1$ four times with the right angles($\mathbf{x_m} = R_y0$) and in the right order alternated with step $C2$ the system will be in the state

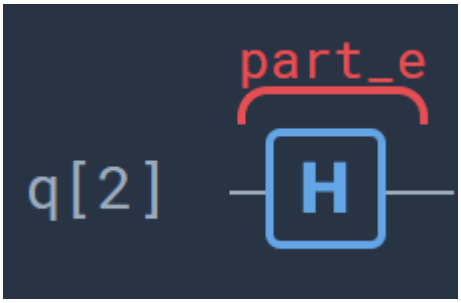
$$\mathcal{D}_C = \frac{1}{\sqrt{8}} \sum_{m=0}^{3} m\Big(0\tilde{\mathbf{x}} + 1\mathbf{x_m}\Big)0$$

- **Part D:** In this part we encode the known labels of the data in the last qubit. This can be done easily using a Toffoli-gate with the controls on the first two qubits and the target on the fifth. Note that this requires only one rotation through m of the labels if we choose the data points such that the two points labeled with $y_m = 1$ are 00 and 11. The combination of twice circuit $D1$ with circuit $D2$ inbetween does the job. The desired state is now produced:

$$\mathcal{D}_D = \frac{1}{\sqrt{8}} \sum_{m=0}^{3} m\Big(0\tilde{\mathbf{x}} + 1\mathbf{x_m}\Big)y_m. \tag{1.16}$$



- **Part E:** In this part the actual distance-based classifier part of the algorithm is done. This part is independent of number of data points, which is precisely the strength of this quantum classifier. This step consists of a simple Hadamard-gate. Results are then obtained by post-processing the measurement results.



Back to Table of Contents

Implementation

We will implement the above algorithm using the projectQ framework. First we need to import some modules and set up the authentication for connecting to the Quantum Inspire API.

[2]:
```
# Imports for authentication with Quantum Inspire API
import os

# Import the projectQ backend from the Quantum Inspire
from quantuminspire.api import QuantumInspireAPI
from quantuminspire.projectq.backend_qx import QIBackend

# Import projectQ
from projectq import MainEngine
from projectq.backends import ResourceCounter
from projectq.ops import CNOT, H, Toffoli, X, CZ, Ry, C
from projectq.setups import restrictedgateset

# Import from the SDK
from quantuminspire.credentials import get_authentication

QI_URL = os.getenv('API_URL', 'https://api.quantum-inspire.com/')


# Remote Quantum-Inspire backend #
authentication = get_authentication()
```

Before we consider the four point classifier, let us first review the 2 point classifier again. Consider the following data points from the Iris flower dataset:

$$\begin{aligned} \psi_{\tilde{x}} &= \quad 0.99990 - 0.00111, && y = 1, \\ \psi_{x_0} &= -0.45830 - 0.88891, && y = 1, \\ \psi_{x_1} &= -0.37280 + 0.92791, && y = 0. \end{aligned} \tag{1.17}$$

The code for this implementation is shown below and treated in detail in the previous notebook.

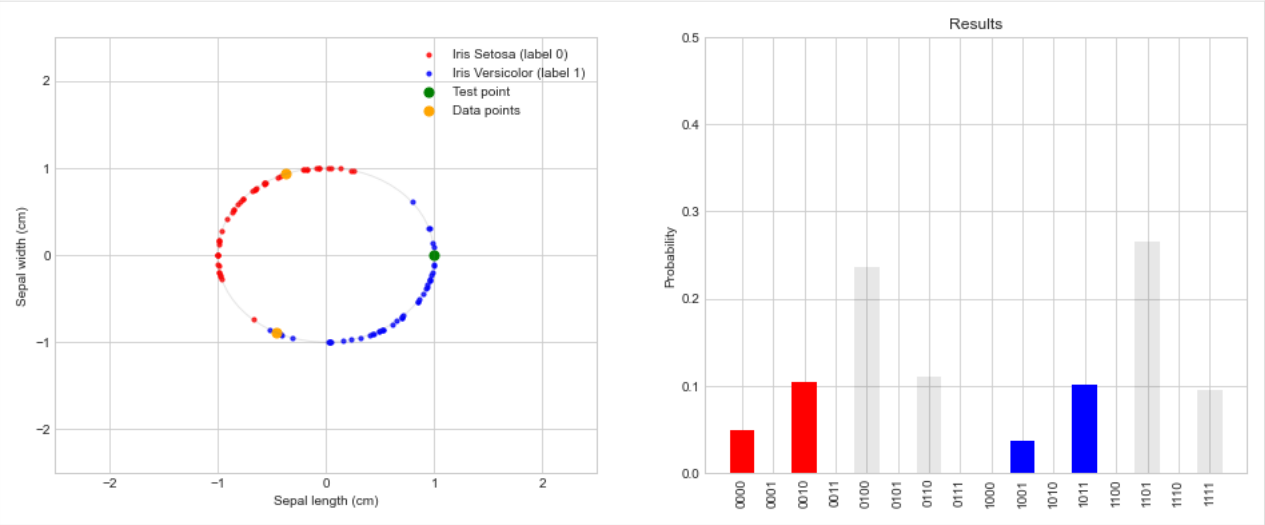
[3]:
```
## 2 points distance-based classifier ##

# Set-up a new connection with qi_backend:
def initialize_qi_backend(project_name = "distance_based_classifier"):
    compiler_engines = restrictedgateset.get_engine_list(one_qubit_gates="any",
                                                         two_qubit_gates=(CNOT, CZ,␣
↪Toffoli))
    compiler_engines.extend([ResourceCounter()])
    qi = QuantumInspireAPI(
        QI_URL,
        authentication,
        project_name = project_name  # Set project name to save projects
    )
    qi_backend = QIBackend(num_runs=1, quantum_inspire_api=qi)# set num_runs = 1 for␣
↪true probability distribution
    qi_engine = MainEngine(backend=qi_backend, engine_list=compiler_engines)
    return qi_backend, qi_engine
```

(continues on next page)

(continued from previous page)

```
qi_backend, qi_engine = initialize_qi_backend("distance_based_classifier_2_points")
# Data points:
x_tilde = [0.9999, -0.0011] # Label 1
x0 = [-0.4583, -0.8889]     # Label 1
x1 = [-0.3728, 0.9279]      # Label 0

# Angles data points:
angle_x_tilde = 2 * acos(x_tilde[0]) * sign(x_tilde[1]) # Label 1
angle_x0 = 2 * acos(x0[0]) * sign(x0[1])                # Label 1
angle_x1 = 2 * acos(x1[0]) * sign(x1[1])                # Label 0

# Quantum circuit:
qubits = qi_engine.allocate_qureg(4)

# part_a
for qubit in qubits[0:2]:
    H | qubit

# part_b
C(Ry(angle_x_tilde), 1) | (qubits[1], qubits[2])
X | qubits[1]

# part_c
C(Ry(angle_x1), 2) | (qubits[0], qubits[1], qubits[2])
X | qubits[0]

# part_d
C(Ry(angle_x0), 2) | (qubits[0], qubits[1], qubits[2])

# part_e
CNOT | (qubits[0], qubits[3])

# part_f
H | qubits[1]

qi_engine.flush()

# Results:
temp_results = qi_backend.get_probabilities(qubits)
print(temp_results)  # Print the results in a dictionary
prob = DataPlotter.plot_data_points(x_tilde,[x0], [x1], temp_results)  # Function to␣
↪plot the data
```

```
{'1101': 0.2657933, '0100': 0.2355381, '0110': 0.1109331, '0010': 0.1043719, '1011': 0.
↪1019124, '1111': 0.0956278, '0000': 0.0491568, '1001': 0.0366663}
```



The classifier predicts the wrong label 0 for the test point with this combination of data points as:

$$0.04 + 0.1043719 > 0.0366663 + 0.1019124 \rightarrow \text{Assign label 0 to } \tilde{y} \tag{1.18}$$

The left figure gives intuition why the classifier fails to predict the correct label. For this specific combination of data points, the test point is closer to the data point with label 0 than to the data point with label 1. As the prediction is based on only these two data points, it is expected to give this wrong prediction. Note, this problem has nothing to do with the quantum computer used for the calculation. The same results would be obtained with a classical distance-based classifier.

By adding two more data points, one for each label, we improve the classifier so that it is better able to give the right label as prediction. Add the following two points to the calculation:

$$\begin{aligned} \psi_{x_2} &= -0.37280 + 0.92791, & y &= 1, \\ \psi_{x_3} &= -0.45830 - 0.88891, & y &= 0. \end{aligned} \tag{1.19}$$

Consider the quantum circuit for the four point classifier below.

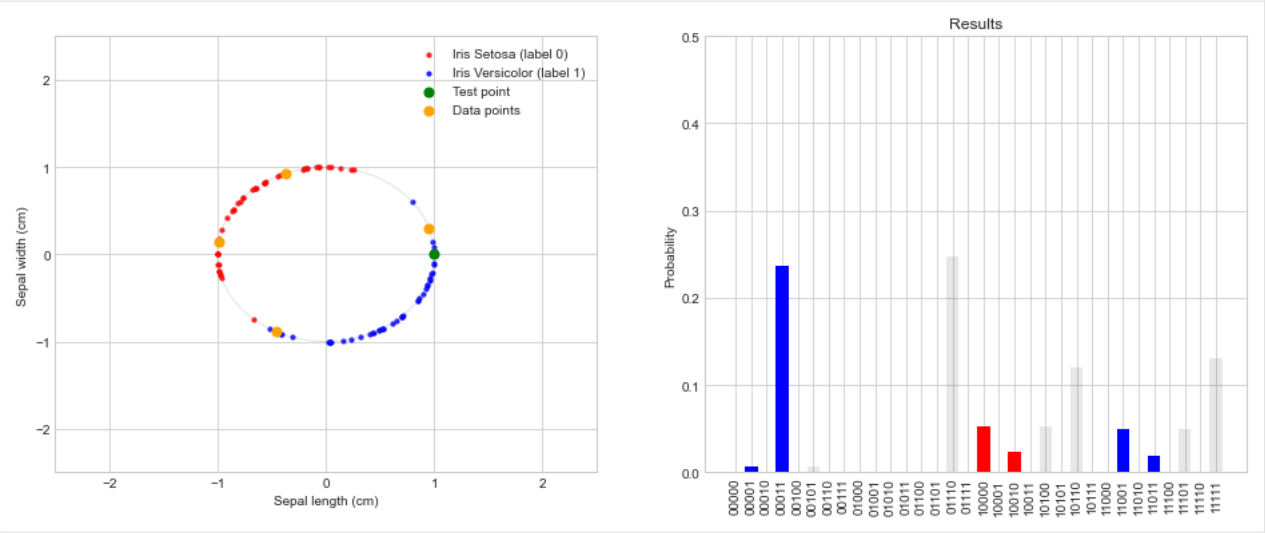
[4]:
```
## 4 points distance-based classifier ##

# Add the 2 new data points:
x_tilde = [0.9999, -0.0011] # Label 1
x0 = [-0.4583, -0.8889]     # Label 1
x1 = [-0.3728, 0.9279]      # Label 0
x2 = [-0.9886, 0.1503]      # Label 0
x3 = [0.9530, 0.3028]       # Label 1


def four_point_distance_based_classifier_circuit(x_tilde, x0, x1, x2, x3):
    # Set-up a new connection with qi_backend:
    qi_backend, qi_engine = initialize_qi_backend("distance_based_classifier_4_points")

    # Angles data points:
    angle_x_tilde = 2 * acos(x_tilde[0]) * sign(x_tilde[1])
    angle_x0 = 2 * acos(x0[0]) * sign(x0[1])   # Label 1
    angle_x1 = 2 * acos(x1[0]) * sign(x1[1])   # Label 0
    angle_x2 = 2 * acos(x2[0]) * sign(x2[1])   # Label 0
```

(continues on next page)

(continued from previous page)

```
    angle_x3 = 2 * acos(x3[0]) * sign(x3[1])   # Label 1

    # Quantum circuit:
    qubits = qi_engine.allocate_qureg(5)

    # part_a
    for qubit in qubits[0:3]:
        H | qubit

    # part_b
    C(Ry(angle_x_tilde), 1) | (qubits[2], qubits[3])
    X | qubits[3]

    # part_c
    for angle in [angle_x0, angle_x1, angle_x2]:
        C(Ry(angle), 3) | (qubits[0], qubits[1], qubits[2], qubits[3]) #C1
        CNOT | (qubits[1], qubits[0]) #C2
        X | qubits[1]
    C(Ry(angle_x3), 3) | (qubits[0], qubits[1], qubits[2], qubits[3]) #C1

    # part_d
    Toffoli | (qubits[0], qubits[1], qubits[4]) #D1
    CNOT | (qubits[1], qubits[0]) #D2
    X | qubits[1]
    Toffoli | (qubits[0], qubits[1], qubits[4]) #D1

    # part_e
    H | qubits[2]

    qi_engine.flush()
    temp_results = qi_backend.get_probabilities(qubits)

    # Results:
    temp_results = qi_backend.get_probabilities(qubits)
    return temp_results


temp_results = four_point_distance_based_classifier_circuit(x_tilde, x0, x1, x2, x3)
print(temp_results, '\n')
prob = DataPlotter.plot_data_points(x_tilde, [x1, x2], [x0, x3], temp_results)  #␣
↪Function to plot the data
```

```
{'01110': 0.2476628, '00011': 0.2373197, '11111': 0.1306251, '10110': 0.1200422, '10000':
↪ 0.0531932, '10100': 0.0531932, '11001': 0.0500852, '11101': 0.0500852, '10010': 0.
↪023571, '11011': 0.019204, '00001': 0.0062575, '00101': 0.0062575, '01000': 0.0011657,
↪'01100': 0.0011657, '00111': 0.000165, '01010': 5.5e-06}
```



Just as we did in previous notebook, we need to count only those outcomes where the third qubit is equal to a 0. That is, we only consider the 16 outcomes in the set:

$$\{00000, 01000, 10000, 11000, 00001, 01001, 10001, 11001, 00010, 01010, 10010, 11010, 00011, 01011, 10011, 11011\}$$

The label $\tilde{y}$ is then given by a majority vote:

$$\begin{aligned} \tilde{y}_0 &= \#\{00000, 01000, 10000, 11000, 00010, 01010, 10010, 11010\} \\ \tilde{y}_1 &= \#\{00001, 01001, 10001, 11001, 00011, 01011, 10011, 11011\} \end{aligned} \tag{1.20}$$

$$\tilde{y} = \begin{cases} -1 & : \tilde{y}_0 > \tilde{y}_1 \\ +1 & : \tilde{y}_1 > \tilde{y}_0 \end{cases} \tag{1.21}$$

[5]:
```
def summarize_results_4_points(prob, display=1):
    def get_label_0_1(n, index_0, index_label):
        # n = number of qubits excluding ancillas
        # index_0 = index of bit that should always be zero to continue
        # index_label = index of bit that should be used to classify in label 0 and␣
↪label 1
        index_0 = [i for i in range(2**n) if (format(i, '#0{0}b'.format(2+n))[2+index_0]␣
↪== '0')]
        label0index = [i for i in index_0 if (format(i, '#0{0}b'.format(2+n))[2+index_
↪label] == '0')]
        label1index = [i for i in index_0 if (format(i, '#0{0}b'.format(2+n))[2+index_
↪label] == '1')]
        return label0index, label1index
    label0indx, label1indx = get_label_0_1(5, 2, 4)
    sum_label0 = 0
    sum_label1 = 0

    for indx in label0indx:
        sum_label0 += prob[indx]
    for indx in label1indx:
        sum_label1 += prob[indx]
```

(continues on next page)

(continued from previous page)

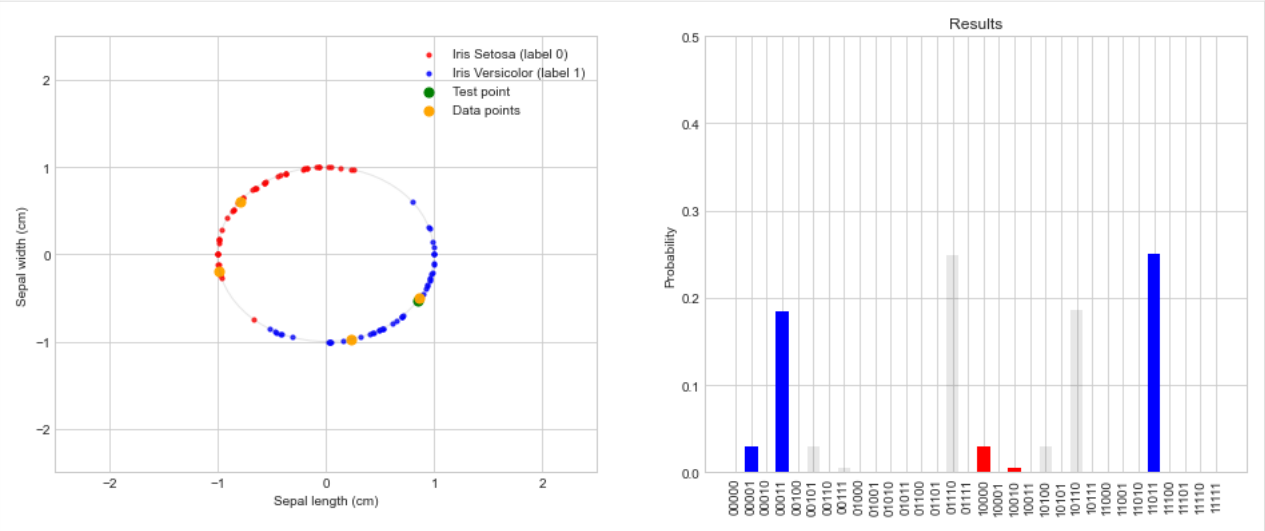
```
    def y_tilde():
        if sum_label0 > sum_label1:
            return 0, ">"
        elif sum_label0 < sum_label1:
            return 1, "<"
        else:
            return "undefined", "="
    y_tilde_res, sign = y_tilde()
    if display:
        print("The sum of the events with label 0 is: {}".format(sum_label0))
        print("The sum of the events with label 1 is: {}".format(sum_label1))
        print("The label for y_tilde is: {} because sum_label0 {} sum_label1".format(y_
↪tilde_res, sign))
    return y_tilde_res


summarize_results_4_points(prob);
```

```
The sum of the events with label 0 is: 0.0779354
The sum of the events with label 1 is: 0.3128664
The label for y_tilde is: 1 because sum_label0 < sum_label1
```

By adding these two points we see that the classifier now gives the correct label for the test data. To see how well this 4-point distance-based classifier performs we also apply it to randomly selected training and test data.

[6]:
```
# Get random data
data_label_0, data_label_1, x_tilde, random_label = DataPlotter.grab_random_data(size=4)
x1, x2 = data_label_0
x0, x3 = data_label_1

temp_results = four_point_distance_based_classifier_circuit(x_tilde, x0, x1, x2, x3)
prob = DataPlotter.plot_data_points(x_tilde, data_label_0, data_label_1, temp_results)
↪# Function to plot the data
summarize_results_4_points(prob);
```

```
The sum of the events with label 0 is: 0.0351141
The sum of the events with label 1 is: 0.4649043
The label for y_tilde is: 1 because sum_label0 < sum_label1
```



Conclusion and further work

In general, a distance-based classifier gets better in predicting classes once more data is included. In this notebook we showed how to extend the previous algorithm to 4 data points. We saw an example where the 2-point distance-based classifier fails to predict the right label, while, when extended to four data points, it managed to classify the test point correctly. This is what we expect classically, however, the key takeaway here is that the quantum algorithm itself (step f) did not change. It is independent of the size of dataset and it is from this that we can expect huge speedups.

Extending the classifier to more data points is now analogus. Extending the classifier to 8 data points will be a nice challenge for the reader to test their understanding, a solution is given below.

In this notebook we used the projectQ backend to generate the quantum algorithm. Note that for controlled gate operations in generall ancillary qubits are required. ProjectQ code does not minimize the amount of ancillary qubits needed in the algorithm, this could be improved.

The next notebook in this series will look at how to include more than two features for the data.

Back to Table of Contents

References

- Book: Schuld and Petruccione, Supervised learning with Quantum computers, 2018
- Article: Schuld, Fingerhuth and Petruccione, Implementing a distance-based classifier with a quantum interference circuit, 2017

Solution for 8 data points

[7]:
```
## 8 points distance-based classifier ##
### Note: Far from qubit optimal solution

# Get data:
data_label_0, data_label_1, x_tilde, random_label = DataPlotter.grab_random_data(size=8)


def eight_point_distance_based_classifier_circuit(x_tilde, data_label_0, data_label_1):
    # Set-up a new connection with qi_backend:
    qi_backend, qi_engine = initialize_qi_backend("distance_based_classifier_8_points")

    # Angles data points:
    angle_label0 = []
    angle_label1 = []
    for data_point in data_label_0:
        angle_label0.append(2 * acos(data_point[0]) * sign(data_point[1]))
    for data_point in data_label_1:
        angle_label1.append(2 * acos(data_point[0]) * sign(data_point[1]))
    angle_x_tilde = 2 * acos(x_tilde[0]) * sign(x_tilde[1])

    # Quantum circuit:
    qubits = qi_engine.allocate_qureg(9)  # 6 qubits + 3 ancillary qubits for CCCCRy␣
↪gates

    # part_a
    for qubit in qubits[0:4]:
        H | qubit

    # part_b
    C(Ry(angle_x_tilde), 1) | (qubits[3], qubits[4])
    X | qubits[3]

    # part_c
    for angle in angle_label1:
        # Build CCCCRy gate from 3 ancillary, Toffoli, and a CRy gate.
        Toffoli | (qubits[0], qubits[1], qubits[6])
        Toffoli | (qubits[2], qubits[6], qubits[7])
        Toffoli | (qubits[3], qubits[7], qubits[8])
        C(Ry(angle), 1) | (qubits[8], qubits[4])
        Toffoli | (qubits[3], qubits[7], qubits[8])
        # Set y_m label conditioned on first three qubits being 1, (don't include 4th␣
↪qubit)
        CNOT | (qubits[7], qubits[5])
```

(continues on next page)

(continued from previous page)

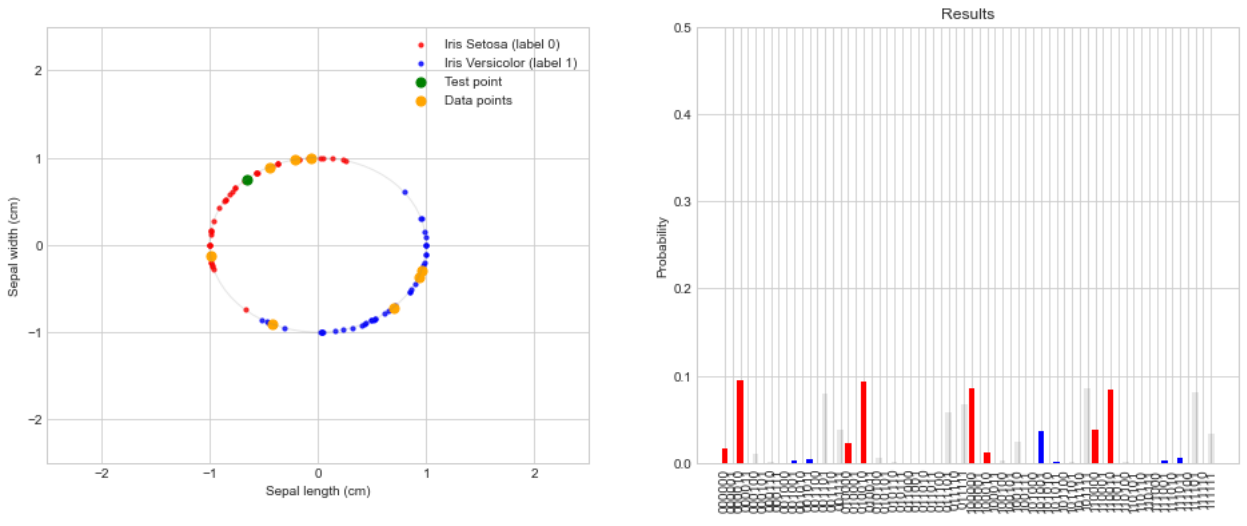
```
        Toffoli | (qubits[2], qubits[6], qubits[7])
        Toffoli | (qubits[0], qubits[1], qubits[6])

        Toffoli | (qubits[0], qubits[1], qubits[2])
        CNOT | (qubits[0], qubits[1])
        X | qubits[0]

    for angle in angle_label0:
        # Build CCCCRy gate from 3 ancillary, Toffoli, and a CRy gate.
        Toffoli | (qubits[0], qubits[1], qubits[6])
        Toffoli | (qubits[2], qubits[6], qubits[7])
        Toffoli | (qubits[3], qubits[7], qubits[8])
        C(Ry(angle), 1) | (qubits[8], qubits[4])
        Toffoli | (qubits[3], qubits[7], qubits[8])
        Toffoli | (qubits[2], qubits[6], qubits[7])
        Toffoli | (qubits[0], qubits[1], qubits[6])

        Toffoli | (qubits[0], qubits[1], qubits[2])
        CNOT | (qubits[0], qubits[1])
        X | qubits[0]

    # part_d
    H | qubits[3]

    qi_engine.flush()
    temp_results = qi_backend.get_probabilities(qubits)

    # Results:
    temp_results = qi_backend.get_probabilities(qubits)
    return temp_results


temp_results = eight_point_distance_based_classifier_circuit(x_tilde, data_label_0, data_
↪label_1)


def strip_ancillary_qubits(results):
    histogram_results = {}
    for k, v in results.items():
        histogram_results[k[:-3]] = v  # Strip ancillary qubits
    return histogram_results


histogram_results = strip_ancillary_qubits(temp_results)
print(histogram_results)


def summarize_results_8_points(histogram_results):
    sum_label0 = 0
    sum_label1 = 0
    for key, value in histogram_results.items():
        if key[3] == "0":
```

(continues on next page)

(continued from previous page)

```
            if key[-1] == "0":
                sum_label0 += value
            else:
                sum_label1 += value
    print("Sum Iris Setosa (red, label 0): ", sum_label0)
    print("Sum Iris Versicolor (blue, label 1): ", sum_label1)


summarize_results_8_points(histogram_results)
# Function to plot the data
prob = DataPlotter.plot_data_points(x_tilde, data_label_0, data_label_1, histogram_
↪results)
```

```
{'000010': 0.095694, '010010': 0.0935606, '101111': 0.0860989, '100000': 0.0852288,
↪'110010': 0.0845656, '111101': 0.0814454, '001101': 0.0790893, '011111': 0.0674479,
↪'011101': 0.0574683, '001111': 0.0388524, '110000': 0.0384513, '101001': 0.0363496,
↪'111111': 0.0342961, '100110': 0.023929, '010000': 0.0234737, '000000': 0.0164274,
↪'100010': 0.0123691, '000100': 0.0109914, '111011': 0.0065148, '010100': 0.0063677,
↪'001011': 0.0047329, '100100': 0.0034728, '111001': 0.0027434, '001001': 0.002325,
↪'000110': 0.0018869, '101101': 0.0017938, '010110': 0.0015976, '110100': 0.001363,
↪'101011': 0.0007573, '110110': 0.0006198, '011001': 4.51e-05, '011011': 3.84e-05}
Sum Iris Setosa (red, label 0):  0.4497705
Sum Iris Versicolor (blue, label 1):  0.5535065000000000005
```




```
[ ]:
```

A Quantum distance-based classifier (part 3)

Robert Wezeman, TNO

Table of Contents

[1]:
```
## Import external python file
from data_plotter import DataPlotter  # for easier plotting
DataPlotter = DataPlotter()

# Import math functions
import numpy as np
from math import acos, atan, sqrt
from numpy import sign
```

Introduction

This notebook is the third in the series on the quantum distance-based classifier. In the first notebook, we looked at how to build a distance-based classifier with two data points, each having two features. In the second notebook, we looked at how to increase the amount of data points. In this notebook we will look at how to increase the amount of features.

Back to Table of Contents

Problem

We repeat the problem description from the previous notebooks. We define the following binary classification problem: Given the data set

$$\mathcal{D} = \Big\{(\mathbf{x}_1, y_1), \dots (\mathbf{x}_M, y_M)\Big\},$$

consisting of M data points $x_i \in \mathbb{R}^n$ and corresponding labels $y_i \in \{-1, 1\}$, give a prediction for the label $\tilde{y}$ corresponding to an unlabeled data point $\tilde{\mathbf{x}}$. The classifier we shall implement with our quantum circuit is a distance-based classifier and is given by

$$\tilde{y} = \text{sgn}\left(\sum_{m=0}^{M-1} y_m \left[1 - \frac{1}{4M}\left|\tilde{\mathbf{x}} - \mathbf{x}_m\right|^2\right]\right). \tag{1}$$ (1.22)

This is a typical M-nearest-neighbor model where each data point is given a weight related to the distance measure. To implement this classifier on a quantum computer we use amplitude encoding. Details are found in the previous notebooks.

Back to Contents

Theory

In the previous notebooks we encoded the data, each having two features, with a simple $R_y(\theta)$ rotation. The angle for this rotation was chosen such that it rotated the state 0 to the desired state $\mathbf{x}$ corresponding to the data. If we want to include more features for our data points we need to generalize this rotation. Instead of a simple rotation, we now need a combination of gates such that it maps $0\ldots0 \mapsto \mathbf{x} = \sum_i a_i i$, where i is the i^{th} entry of the computational basis $\{0\ldots0, \ldots, 1\ldots1\}$. Again we only work with normalised data, meaning $\sum_i |a_i|^2 = 1$. The general procedure how to initialize a state to an arbitrary superposed state can be found in the article by Long and Sun. In this notebook we will consider how to implement their scheme for 2 qubits, that is up to 4 features:

$$00 \mapsto a_{00}00 + a_{01}01 + a_{10}10 + a_{11}11 \tag{1.23}$$

For the implementation we closely follow the reference and use single bit rotation gates $U(\theta)$ defined by

$$U(\theta) = \begin{pmatrix} \cos(\theta) & \sin(\theta) \\ \sin(\theta) & -\cos(\theta) \end{pmatrix} = R_y(2\theta) \cdot Z \tag{1.24}$$

and controlled versions of it. Because we will only act with these gates on 0 we can even drop the Z gate.

For two qubits the scheme consists of three steps: 1. Apply a bit rotation $U(\alpha_1)$ on the first qubit:

$$U(\alpha_1)0 \otimes 0 = \sqrt{{a_{00}}^2 + {a_{01}}^2}00 + \sqrt{{a_{10}}^2 + {a_{11}}^2}10, \tag{1.25}$$

where α_1 is given by

$$\alpha_1 = \arctan\left(\sqrt{\frac{{a_{10}}^2 + {a_{11}}^2}{{a_{00}}^2 + {a_{01}}^2}}\right) \tag{1.26}$$

2. Next, apply a controlled-rotation $U(\alpha_2)$ on the second qubit, with as control the first qubit being 0. Choose α_2 such that:
$$\cos(\alpha_2) = \frac{a_{00}}{\sqrt{{a_{00}}^2 + {a_{01}}^2}}, \qquad \sin(\alpha_2) = \frac{a_{01}}{\sqrt{{a_{00}}^2 + {a_{01}}^2}} \tag{1.27}$$

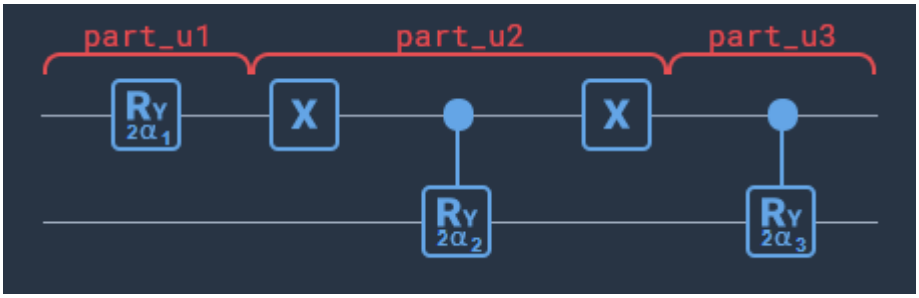
3. Lastly, apply a controlled-rotation $U(\alpha_3)$ on the second qubit, with the first qubit being 1 as control. Choose α_3 such that:
$$\cos(\alpha_3) = \frac{a_{10}}{\sqrt{{a_{10}}^2 + {a_{11}}^2}}, \qquad \sin(\alpha_3) = \frac{a_{11}}{\sqrt{{a_{10}}^2 + {a_{11}}^2}} \tag{1.28}$$

The angles α_2 and α_3 are chosen such that the root terms cancel out, leaving us with the desired result:

$$\begin{aligned} &\sqrt{{a_{00}}^2 + {a_{01}}^2}0 \otimes U(\alpha_2)0 + \sqrt{{a_{10}}^2 + {a_{11}}^2}1 \otimes U(\alpha_3)0 \\ &\quad = a_{11}00 + a_{01}01 + a_{10}10 + a_{11}11 \end{aligned} \tag{1.29}$$

Note: this circuit makes it also possible to encode 3 features by simply setting $a_{11} = 0$.

The circuit looks something like this:



Back to Table of Contents

Algorithm for the arbitrary state preparation

In this notebook we will work with the Qiskit backend for the quantum inspire. Let us first take a closer look at the state preparation part of the circuit used to preparing an arbitrary state. The following code loads the scaled and normalised data of the Iris set containing all 4 features. Let us consider the first point in this set

[2]:
```
# Load scaled and normalised data
iris_setosa_normalised, iris_versicolor_normalised = DataPlotter.load_data(max_
↪features=4)
first_data_point = [feature[0] for feature in iris_setosa_normalised]
first_data_point
```

[2]:
```
[-0.32695550305984783,
 0.4736868636179572,
 -0.5699845991812187,
 -0.5863773622428564]
```

We want to build the circuit such that:

$$00 \mapsto -0.327000 + 0.473701 - 0.570010 - 0.586411 \tag{1.30}$$

[3]:
```
a00, a01, a10, a11 = first_data_point
alpha1 = atan(sqrt((a10**2  + a11**2) / (a00**2  + a01**2)))
alpha2 = np.arctan2(a01, a00)
alpha3 = np.arctan2(a11, a10)
```

As always the first step is to set up a connection with the Quantum Inspire:

[4]:
```
import os

from qiskit.circuit import QuantumRegister, ClassicalRegister, QuantumCircuit
from qiskit.tools.visualization import plot_histogram
from qiskit import execute

from quantuminspire.api import QuantumInspireAPI
from quantuminspire.credentials import get_authentication
from quantuminspire.qiskit import QI

QI_URL = os.getenv('API_URL', 'https://api.quantum-inspire.com/')
```

(continues on next page)

(continued from previous page)

```
authentication = get_authentication()
# Temporary alternative so that we can give our project a name:
QI._api = QuantumInspireAPI(QI_URL, authentication, project_name="Distance-based␣
↪Classifier more features (NEW)")

# Alternative:
# authentication = [email, password]
# QI.set_authentication_details(*authentication)

qi_backend = QI.get_backend('QX single-node simulator')
```

We can now construct a function which builds the quantum circuit as discussed above. Note that Qiskit contains the general unitary gates $u3(\theta, \phi, \lambda)$ which are related to our definition of $U(\alpha)$ by:

$$U(\alpha) = u3(2\alpha, 0, \pi).$$

Unfortunately, these $u3$ gates are not yet implemented on the quantum inspire backend and thus we need to make use of regular R_y rotations and CNOT gates.

[5]:
```
def cRy(q, angle, ctrlidx, idx):
    if len(q) < 2:
        raise ValueError("Error, len quantum register must at least be 2.")
    circuit = QuantumCircuit(q)
    half_angle = angle / 2

    circuit.cx(q[ctrlidx], q[idx])
    circuit.ry(-half_angle, q[idx])
    circuit.cx(q[ctrlidx], q[idx])
    circuit.ry(half_angle, q[idx])
    return circuit


def features_encoding(q, alpha, idx1, idx2):
    if len(q) < 2:
        raise ValueError("Error, len quantum register must at least be 2.")
    # Alternative use u3(2 * alpha, 0, pi) and cu3(2 * alpha, 0, pi) gates but not yet␣
↪implemented on Quantum inspire
    alpha1, alpha2, alpha3 = alpha

    circuit = QuantumCircuit(q)

    # step 1.
    circuit.ry(2 * alpha1, q[idx1])

    # # step 2.
    circuit.x(q[idx1])
    circuit = circuit.compose(cRy(q, 2 * alpha2, idx1, idx2))
    circuit.x(q[idx1])

    # step 3.
    circuit = circuit.compose(cRy(q, 2 * alpha3, idx1, idx2))
    return circuit
```

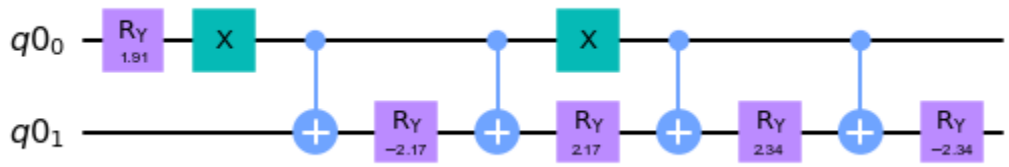
(continues on next page)

(continued from previous page)

```
def measurement(q):
    # TO DO: Rewrite as loop over b to avoid measurement of ancillary qubits
    circuit_size = len(q)
    b = ClassicalRegister(circuit_size)
    meas = QuantumCircuit(q, b)
    meas.measure(q, b)
    return meas
```

[6]:
```
q = QuantumRegister(2)
circuit = features_encoding(q, [alpha1, alpha2, alpha3], 0, 1)
circuit.draw(output='mpl')
```

[6]:



[7]:
```
#Initialize quantum register:
q = QuantumRegister(2)
meas = measurement(q)

# Build circuit:
circuit = features_encoding(q, [alpha1, alpha2, alpha3], 0, 1)
qc = meas.compose(circuit, front=True)


# Execute the circuit:
def execute_circuit(circuit, shots=1):
    qi_job = execute(qc, backend=qi_backend, shots=shots)

    # Temporary needed to fix naming the project:
    project = next((project for project in QI._api.get_projects()
                    if project['name'] == QI._api.project_name), None)
    if project is not None:
        qi_job._job_id = str(project['id'])

    # Results of the job:
    qi_result = qi_job.result()

    # Select the results of the circuit and print results in a dict
    probabilities = qi_result.get_probabilities(qc)
    return probabilities


def create_histogram_results(probabilities, number_of_bits=2):
    histogram_results = {}
```

(continues on next page)

(continued from previous page)

```
    for k, v in probabilities.items():
        # Reversed order of the bin strings to be consistent
        histogram_results[k[::-1]] = v
    return histogram_results


# Display results:
probabilities = execute_circuit(qc)
create_histogram_results(probabilities)
```

[7]:
```
{'00': 0.1068999, '10': 0.3248824, '01': 0.2243791, '11': 0.3438387}
```

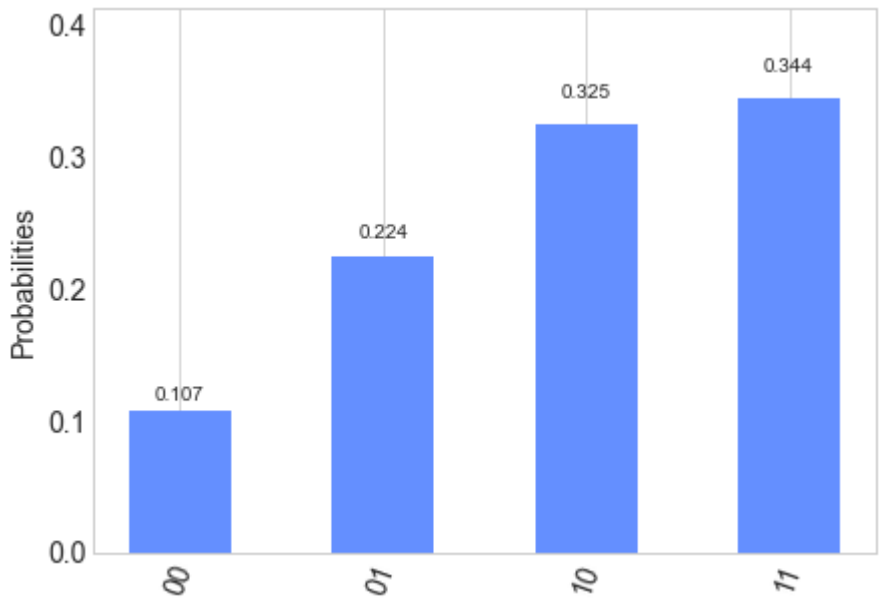
We can compare these results with what we expected if we measure the state $\psi = a_{00}00 + a_{01}01 + a_{10}10 + a_{11}11$ and obtain the result X

Outcome x	$Prob(X = x)$
00	${a_{00}}^2$
01	${a_{01}}^2$
10	${a_{10}}^2$
11	${a_{11}}^2$

[8]:
```
# Desired outcomes for the test point:
desired_outcomes = [coefficient**2 for coefficient in [a00, a01, a10, a11]]
print('Desired values: ', desired_outcomes)
plot_histogram(create_histogram_results(probabilities))
```

```
Desired values:  [0.10689990098111816, 0.2243792447642172, 0.3248824433037745, 0.
↪34383841095089007]
```

[8]:



In the next section we use this feature encoding schema in the distance-based classifier for 2 data points, each having 4 features. In the algorithm however, feature encoding is done in a controlled fashion, with the index qubit as control. Therefore we will also need the same circuit as above but transformed to a controlled version. This can be done by

replacing every gate by a controlled equivalent.

[9]:
```
def ccRy(q, angle, ctrlidx1, ctrlidx2, idx):
    if len(q) < 3:
        raise ValueError("Error, len quantum register must at least be 3.")
    circuit = QuantumCircuit(q)
    quarter_angle = angle / 4

    circuit.ccx(q[ctrlidx1], q[ctrlidx2], q[idx])
    circuit.cx(q[ctrlidx2], q[idx])
    circuit.ry(quarter_angle, q[idx])
    circuit.cx(q[ctrlidx2], q[idx])
    circuit.ry(- quarter_angle, q[idx])

    circuit.ccx(q[ctrlidx1], q[ctrlidx2], q[idx])
    circuit.cx(q[ctrlidx2], q[idx])
    circuit.ry(-quarter_angle, q[idx])
    circuit.cx(q[ctrlidx2], q[idx])
    circuit.ry(quarter_angle, q[idx])
    return circuit


def c_features_encoding(q, alpha, ctrlidx, idx1, idx2):
    if len(q) < 3:
        raise ValueError("Error, len quantum register must at least be 3.")
    alpha1, alpha2, alpha3 = alpha
    circuit = QuantumCircuit(q)

    # step 1.
    circuit = circuit.compose(cRy(q, 2 * alpha1, ctrlidx, idx1))  # old: ry(2 * alpha1,␣
↪q[idx1])

    # # step 2.
    circuit.cx(q[ctrlidx], q[idx1])  # old: x(q[idx1])
    circuit = circuit.compose(ccRy(q, 2 * alpha2, idx1, ctrlidx, idx2))  # old: cRy gates
    circuit.cx(q[ctrlidx], q[idx1])  # old: x(q[idx1])

    # step 3.
    circuit = circuit.compose(ccRy(q, 2 * alpha3, idx1, ctrlidx, idx2))  # old: cRy gates
    return circuit
```

Back to Table of Contents

Implementation of the distance-based classifier

We first consider the case with 2 data points and just 2 features, randomly chosen from the Iris data set. This so that we can later compare the results of the distance-based classifier with 2 features to the implementation with 4 features.

[10]:
```
def two_features_classifier(q, x_tilde, x0, x1):
    if len(q) != 4:
        raise ValueError("Error, len quantum register must be 4.")
    circuit = QuantumCircuit(q)
```

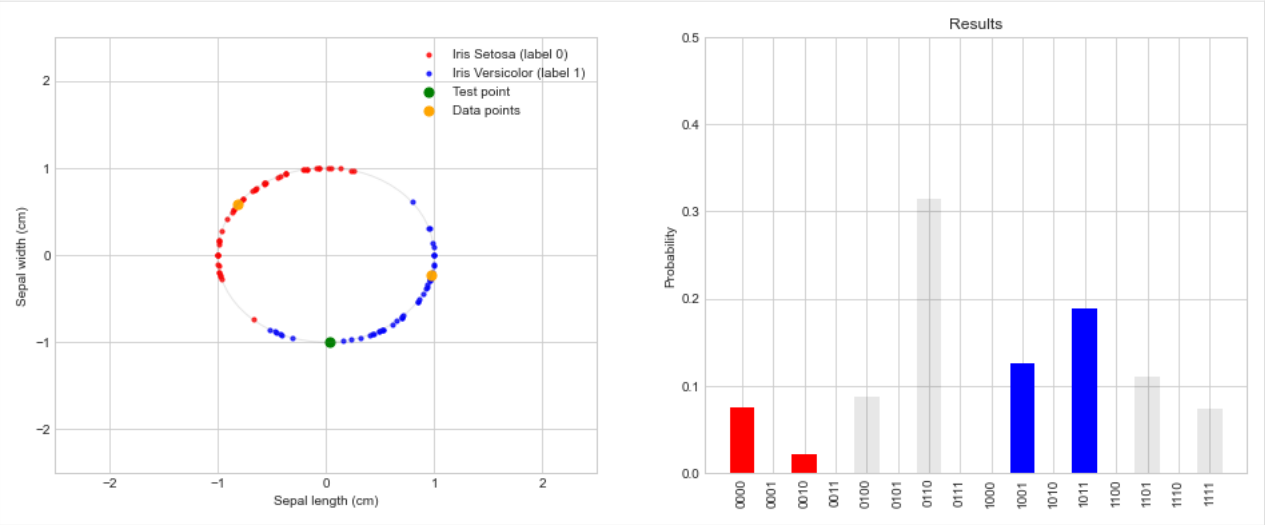
(continues on next page)

(continued from previous page)

```
    # Angles data points:
    angle_x_tilde = 2 * acos(x_tilde[0]) * sign(x_tilde[1]) # Label ?
    angle_x0 = 2 * acos(x0[0]) * sign(x0[1])                # Label 0
    angle_x1 = 2 * acos(x1[0]) * sign(x1[1])                # Label 1

    # part_a:
    for i in range(2):
        circuit.h(q[i])

    # part_b:
    circuit = circuit.compose(cRy(q, angle_x_tilde, 1, 2))
    circuit.x(q[1])

    # part_c:
    circuit = circuit.compose(ccRy(q, angle_x0, 1, 0, 2))
    circuit.x(q[0])

    # part_d:
    circuit = circuit.compose(ccRy(q, angle_x1, 1, 0, 2))

    # part_e:
    circuit.cx(q[0], q[3])

    # part_f:
    circuit.h(q[1]);
    return circuit
```

The above algorithm is the same implementation of the algorithm as we did in the first notebook. The following code runs the algorithm on randomly selected data from the iris set.

[11]:
```
# Get 2 random data points with 2 features.
data_label0, data_label1, x_tilde, random_label = DataPlotter.grab_random_data(size=2,␣
↪features=2)

#Initialize quantum register:
q = QuantumRegister(4)
meas = measurement(q)

# Initialize circuit:
circuit = two_features_classifier(q, x_tilde, data_label0[0], data_label1[0])
qc = meas.compose(circuit, front=True)

# Execute the circuit:
probabilities = execute_circuit(qc)

# Display the results:
histogram_results = create_histogram_results(probabilities, number_of_bits=4)
DataPlotter.plot_data_points(x_tilde, data_label0, data_label1, histogram_results);  #␣
↪Function to plot the data
```



Below the implementation for four features is given. Note that the structure of the algorithm is similar to the case with two features. We use one ancillary qubit for the controlled encoding of the features.

[12]:
```
def four_features_classifier(q, x_tilde, x0, x1):
    if len(q) != 6:
        raise ValueError("Error, len quantum register must be 5 + 1 ancillary qubit.")
    circuit = QuantumCircuit(q)

    def get_alpha(data_point):
        a00, a01, a10, a11 = data_point
        alpha1 = atan(sqrt((a10**2  + a11**2) / (a00**2  + a01**2)))
        alpha2 = np.arctan2(a01, a00)
        alpha3 = np.arctan2(a11, a10)
        return [alpha1, alpha2, alpha3]

    # part_a:
    for i in range(2):
        circuit.h(q[i])

    # part_b:
    alpha = get_alpha(x_tilde)
    circuit = circuit.compose(c_features_encoding(q, alpha, 1, 2, 3))
    circuit.x(q[1])

    # part_c:
    # Use ancillary qubit + c_features_encoding for cc_features_encoding
    circuit.ccx(q[0], q[1], q[5])
    alpha = get_alpha(x0)
    circuit = circuit.compose(c_features_encoding(q, alpha, 5, 2, 3))
    circuit.ccx(q[0], q[1], q[5])
    circuit.x(q[0])

    # part_d:
    # Use ancillary qubit + c_features_encoding for cc_features_encoding
    circuit.ccx(q[0], q[1], q[5])
    alpha = get_alpha(x1)
```

(continues on next page)

(continued from previous page)

```
    circuit = circuit.compose(c_features_encoding(q, alpha, 5, 2, 3))
    circuit.ccx(q[0], q[1], q[5])

    # part_e:
    circuit.cx(q[0], q[4])

    # part_f:
    circuit.h(q[1])

    return circuit
```

The following code runs the algorithm for 2+1 random data points with 4 features each.

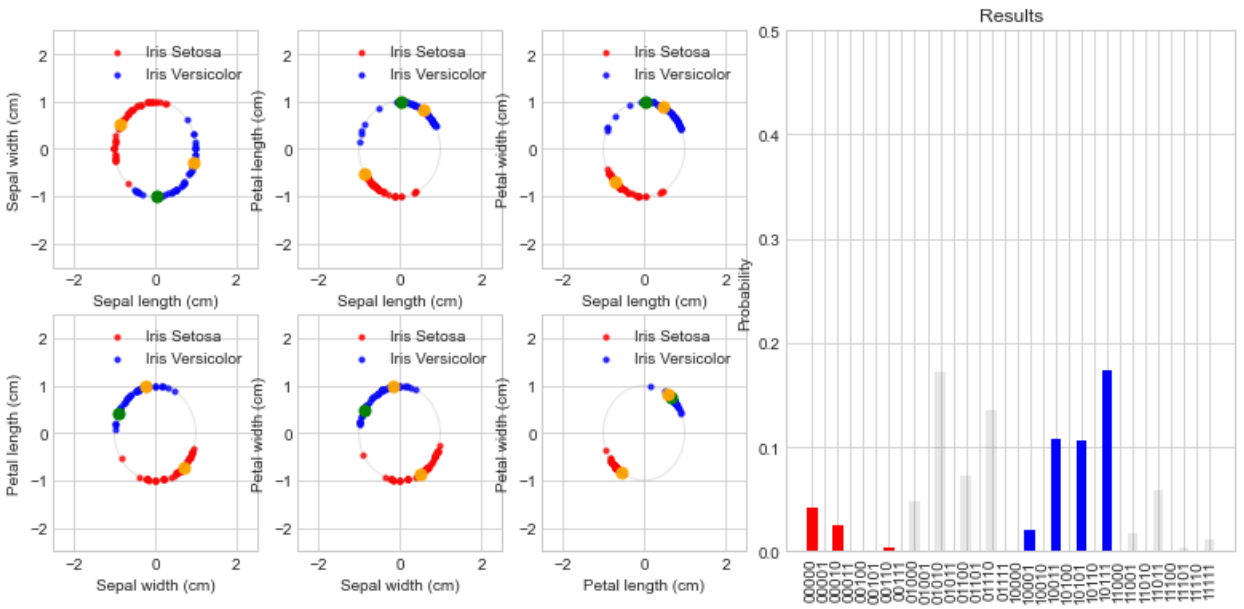
[13]:
```
# Get 2 random data points with 4 features.
data_label0, data_label1, x_tilde, random_label = DataPlotter.grab_random_data(size=2,␣
↪features=4)

#Initialize quantum register:
q = QuantumRegister(6)
meas = measurement(q)

# Initialize circuit:
circuit = four_features_classifier(q, x_tilde, data_label0[0], data_label1[0])
qc = meas.compose(circuit, front=True)

# Execute the circuit:
probabilities = execute_circuit(qc)

# Display the results:
histogram_results = create_histogram_results(probabilities, number_of_bits=6)


def strip_ancillary_qubit(histogram_results):
    new_histogram_results = {}
    for k, v in histogram_results.items():
        new_histogram_results[k[:-1]] = v  # Strip ancillary qubit
    return new_histogram_results


histogram_results = strip_ancillary_qubit(histogram_results)


def summarize_results_4_features(histogram_results):
    sum_label0 = 0
    sum_label1 = 0
    for key, value in histogram_results.items():
        if key[1] == "0":
            if key[-1] == "0":
                sum_label0 += value
            else:
                sum_label1 += value
    print("Sum Iris Setosa (red, label 0): ", sum_label0)
```

(continues on next page)

(continued from previous page)

```
    print("Sum Iris Versicolor (blue, label 1): ", sum_label1)


summarize_results_4_features(histogram_results)
print("Random label is: ", random_label)
print("Prediction by true classifier: ", DataPlotter.true_classifier(data_label0, data_
↪label1, x_tilde))

# Plot results:
DataPlotter.plot_data_points_multiple_features(
    data_label0, data_label1, x_tilde, random_label, histogram_results) # Plot features␣
↪+ bar plot results
```

```
Sum Iris Setosa (red, label 0):  0.0704467
Sum Iris Versicolor (blue, label 1):  0.4095059
Random label is:  1
Prediction by true classifier:  1
```



In the case of an infinite amount of shots the quantum inspire gives as a result the true probability distribution which coincides with the classical solution of the distance-based classifier. The following table shows the quality of the distance-based classifier depending on the amount of data points and included features. The table contains the percentage of correct predictions for random selected data from the iris set, the results are over a sample of 10.000 runs and can be reproduced using the quality_classifier method of DataPlotter class.

% correct prediction	2 features	3 features	4 features
2 data points	0.9426	0.9870	0.9940
4 data points	0.9735	0.9933	0.9986
8 data points	0.9803	0.9975	0.9998

These results show why one is not only interested in extending the amount of data points but also in including more features for data.

Conclusion and further work

In this notebook we demonstrated how to extended the distance-based classifier to be able to handle data containing up to four features. We saw that the quality of the distance-based classifier improves both by including more data points but also by including data which containing more features.

So far in this notebook series we have only looked at binary classification, the results belong either to the class with a label 0 or to the class with the label 1. For some problems one is interessted in identifying between more than two classes, for example number recognition. A possible next extention for the current classifier is to extend it to being able to classify between more than two labels. This can be done by encoding the label in multiple qubits instead of one qubit.

We have only tested the distance-based classifier on rescaled data from the iris data set, this data set is well classified by the the distance-based classifier. For other data sets this might not necessary be the case. Suppose a different data set which after scaling has different the classes lie in concentric circles, at first glance we do not expect the distance-based classifier to yield good predictions. These problems can possibly be solved by an alternative data pre-processing or by a totally different type of classifier. The task of selecting the right methods for data preprocessing and the corresponding classifier is not a task for the quantum computer but for the data analyst. It will be interessting to see different classifiers implemented on quantum computers in the near future.

Back to Table of Contents

References

- Book: Schuld and Petruccione, Supervised learning with Quantum computers, 2018
- Article: Schuld, Fingerhuth and Petruccione, Implementing a distance-based classifier with a quantum interference circuit, 2017
- Article: Long & Sun: Efficient scheme for initializing a quantum register with an arbitrary superposed state, 2001
- Post: Build an arbitrary (n)-controlled quantum gate

[ ]:

1.2.3 Knowledgebase code examples

Measurement Error Mitigation on Quantum Inspire

Backends

This example is written for the Spin-2 backend. If it is offline or the authentication unsuccessful, the code will fallback to simulation using Qiskit Aer.

What does it do and what is it used for?

For quantum devices, in particular spin-qubits, the measurement error is significant with respect to other sources of errors. We can reduce the effect of measurement errors using measurement error mitigation. This is particularly important when different qubits have different readout fidelities or when there is a cross-talk between the qubit readouts. Reducing the measurement error can be essential for the improvement of the performance of a quantum algorithm.

For more details see Qiskit textbook on measurement error mitigation and Mitiq documentation.

How does it work?

We will be measuring a Bell state first without error mitigation, and then apply the measurement error mitigation tools available in Qiskit to improve the result. We perform the calculations on real hardware (or on the Qiskit QASM simulator when the hardware is not available). Below we will go through this process in detail.

For information on an advanced technique for measurement error mitigation, have a look at the references at the bottom of the notebook.

To install the required dependencies for this notebook, run:

```
pip install qiskit quantuminspire requests
```

Example Code

Imports and definitions

[19]:
```
import numpy as np
import qiskit
import requests
from qiskit import Aer, QuantumCircuit, QuantumRegister, assemble, execute, transpile
from qiskit.ignis.mitigation.measurement import CompleteMeasFitter, complete_meas_cal
from qiskit.providers.aer.noise import NoiseModel, ReadoutError
from qiskit.providers.aer.noise.errors import depolarizing_error, pauli_error
from qiskit.visualization import plot_histogram
from quantuminspire.api import QuantumInspireAPI
from quantuminspire.credentials import get_authentication
from quantuminspire.qiskit import QI
```

[20]:
```
def get_qi_calibration_from_qiskit_result(qi_job, authentication) -> dict:
    """ Return calibration data from a QI job """
    api = QI.get_api()
    project = api.get_project(qi_job.job_id())
    # NOTE: due to limited capabilties of the QI framework this is the latest job,␣
↪perhaps not the actual job
    job = qi_job.get_jobs()[0]
    result = api.get_result_from_job(job["id"])
    c = result["calibration"]
    result = requests.get(c, auth=authentication)
    calibration = result.json()
    return calibration

def spin_2_noise_model(p: float = 0.01):
```

(continues on next page)

(continued from previous page)

```
    """ Define noise model representation for Spin-2 """
    noise_model = NoiseModel()
    error_gate = pauli_error([("X", p), ("I", 1 - p)])

    noise_model.add_all_qubit_quantum_error(error_gate, ["u1", "u2", "u3", "rx", "ry", "x
↪", "y"])

    read_err = ReadoutError([[0.9, 0.1], [0.2, 0.8]])
    noise_model.add_readout_error(read_err, [0])
    read_err = ReadoutError([[0.7, 0.3], [0.25, 0.75]])
    noise_model.add_readout_error(read_err, [1])
    return noise_model

noise_model_spin2 = spin_2_noise_model()
```

Set up connection to QI

We expect the user to have set up the token authentication as described here in order to be able to access the Spin-2 backend in the following cells.

[21]:
```
try:
    authentication = get_authentication()
    QI.set_authentication(authentication)
    qi_backend = QI.get_backend("Spin-2")
except:
    print("QI connection not available, fall-back to simulation.")
    qi_backend = None
```

```
Enter email:

Enter password
........
QI connection not available, fall-back to simulation.
```

[22]:
```
def execute_circuit_spin2(qc, qi_backend, shots=2048):
    """  Execute circuit on Spin-2 with fall-back to simulation """
    sim = False
    if qi_backend is None:
        sim = True
    else:
        api = QI.get_api()
        backend = api.get_backend_type(qi_backend.backend_name)
        if backend["status"] == "OFFLINE":
            print(f"Backend {qi_backend} is offline, fall-back to simulation")
            sim = True

    if sim:
        backend = Aer.get_backend("qasm_simulator")
        result = backend.run(qc, shots=shots, noise_model=noise_model_spin2).result()
        qi_job = None
    else:
```

(continues on next page)

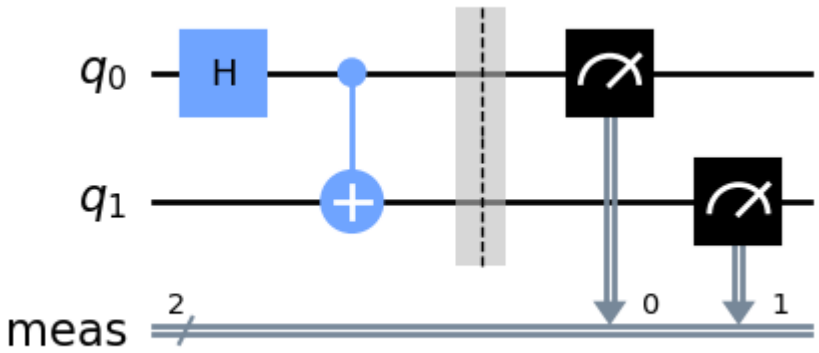
(continued from previous page)

```
        try:
            qi_job = execute(qc, backend=qi_backend, shots=shots)
            result = qi_job.result()
        except Exception as ex:
            print(f"Failed to run on backend {qi_backend}: {ex}, fall-back to simulation
↪")
            backend = Aer.get_backend("qasm_simulator")
            result = backend.run(qc, shots=shots, noise_model=noise_model_spin2).result()
            qi_job = None
    return result, qi_job
```

Error mitigation for Spin-2

Run error mitigation test on a Bell state $(|00\rangle + |11\rangle)/\sqrt{2}$.

[23]:
```
qc = QuantumCircuit(2, name="Bell state")
qc.h(0)
qc.cx(0, 1)
qc.measure_all()
display(qc.draw(output="mpl"))

bell_measurement_result, qi_job = execute_circuit_spin2(qc, qi_backend)
noisy_counts = bell_measurement_result.get_counts()

print(noisy_counts)
```




```
{'01': 278, '11': 656, '10': 441, '00': 673}
```

[24]:
```
if qi_job:
    cr = get_qi_calibration_from_qiskit_result(qi_job, authentication)

    measurement_error_results = cr["parameters"]["system"]["readout_error_calibration"][
↪"qiskit_result"]
    measurement_error_results = qiskit.result.Result.from_dict(measurement_error_results)
```

(continues on next page)

(continued from previous page)

```
    timestamp = cr["parameters"]["system"]["readout_error_calibration"]["timestamp"]
    print(f"Using calibration of {timestamp}")
else:
    measurement_error_results = np.array(
        [
            [0.7421875, 0.12890625, 0.0, 0.0390625],
            [0.11328125, 0.62890625, 0.0, 0.13671875],
            [0.140625, 0.05859375, 1.0, 0.11328125],
            [0.00390625, 0.18359375, 0.0, 0.7109375],
        ]
    )
```

In Qiskit we mitigate the noise by creating a measurement filter object. Then, taking the results from above, we use this to calculate a mitigated set of counts.

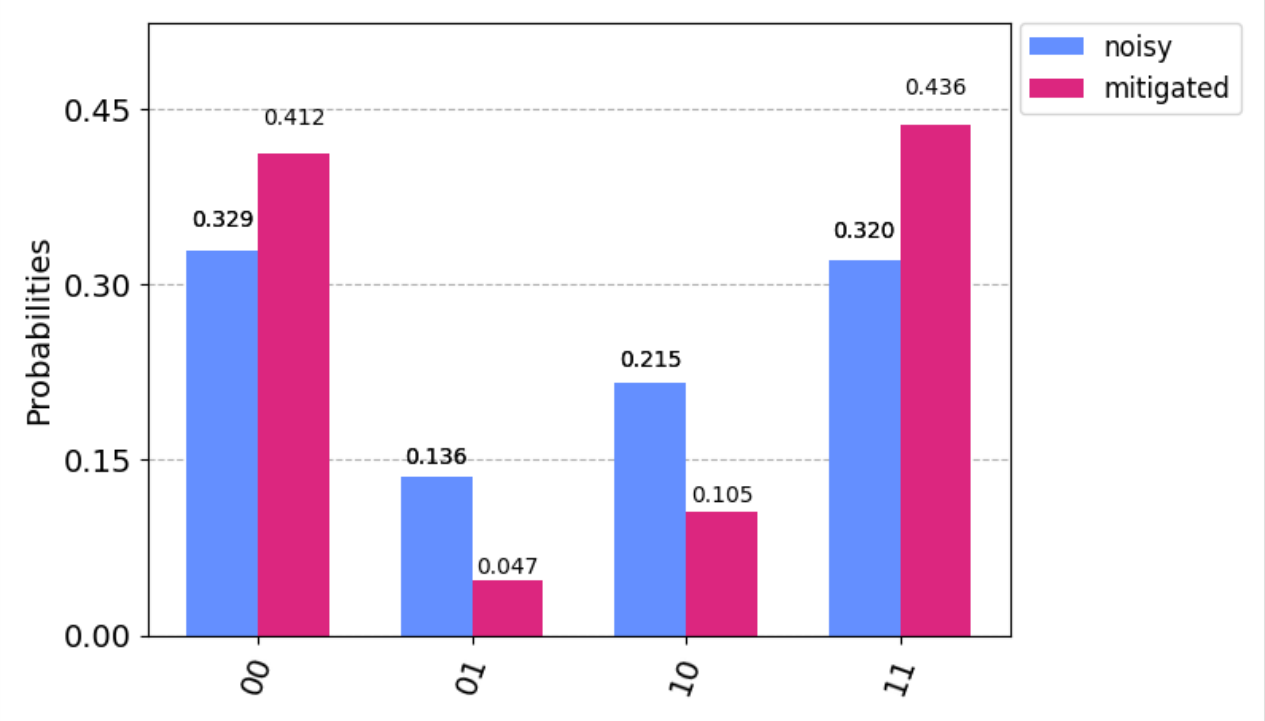
[25]:
```
if isinstance(measurement_error_results, np.ndarray):
    meas_fitter = CompleteMeasFitter(None, state_labels=["00", "01", "10", "11"])
    meas_fitter.cal_matrix = measurement_error_results
else:
    state_labels = ["00", "01", "10", "11"]
    meas_fitter = CompleteMeasFitter(measurement_error_results, state_labels, circlabel="
↪")
```

[26]:
```
meas_filter = meas_fitter.filter

# Results with error mitigation
mitigated_results = meas_filter.apply(bell_measurement_result)
mitigated_counts = mitigated_results.get_counts()
```

To see the results most clearly, let's plot both the noisy and mitigated results.

[27]:
```
plot_histogram([noisy_counts, mitigated_counts], legend=["noisy", "mitigated"])
```

[27]:



Note that the skew between the two qubits has reduced significantly.

Measurement of the error matrix

The process of measurement error mitigation can also be done using the existing tools from Qiskit. This handles the collection of data for the basis states, the construction of the matrices, and the calculation of the inverse. The latter can be done using the pseudoinverse, as we saw above. However, the default is an even more sophisticated method which is using least-squares fitting.

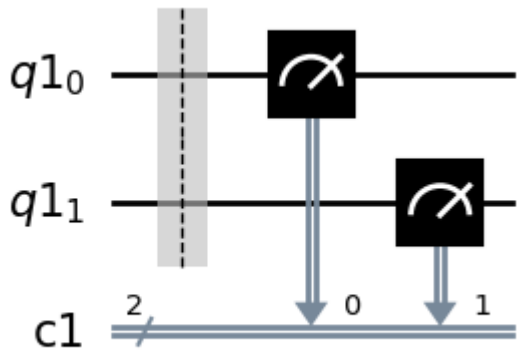
As an example, let's stick with doing error mitigation for a pair of qubits. For this, we define a two-qubit quantum register, and feed it into the function `complete_meas_cal`.

[28]:
```
qr = QuantumRegister(2)
meas_calibs, state_labels = complete_meas_cal(qr=qr, circlabel="")
```

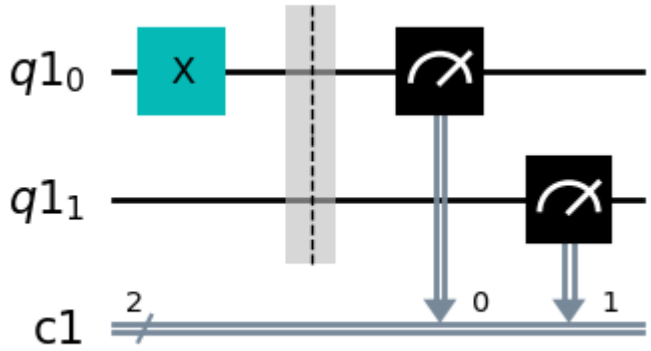
This creates a set of circuits that are used to take measurements for each of the four basis states of two qubits: $|00\rangle$, $|01\rangle$, $|10\rangle$ and $|11\rangle$.

[29]:
```
for circuit in meas_calibs:
    print(f"Circuit {circuit.name}")
    display(circuit.draw(output="mpl"))
```

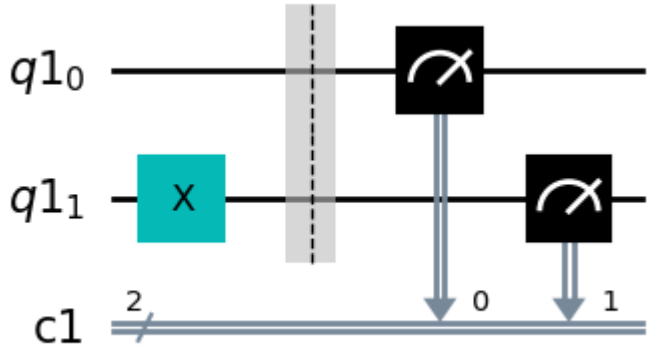
```
Circuit cal_00
```



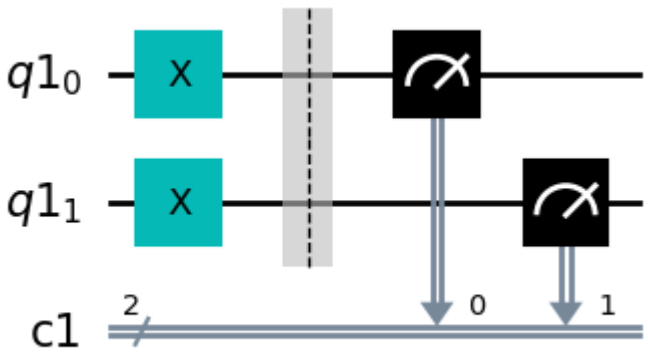
Circuit cal_01



Circuit cal_10



Circuit cal_11

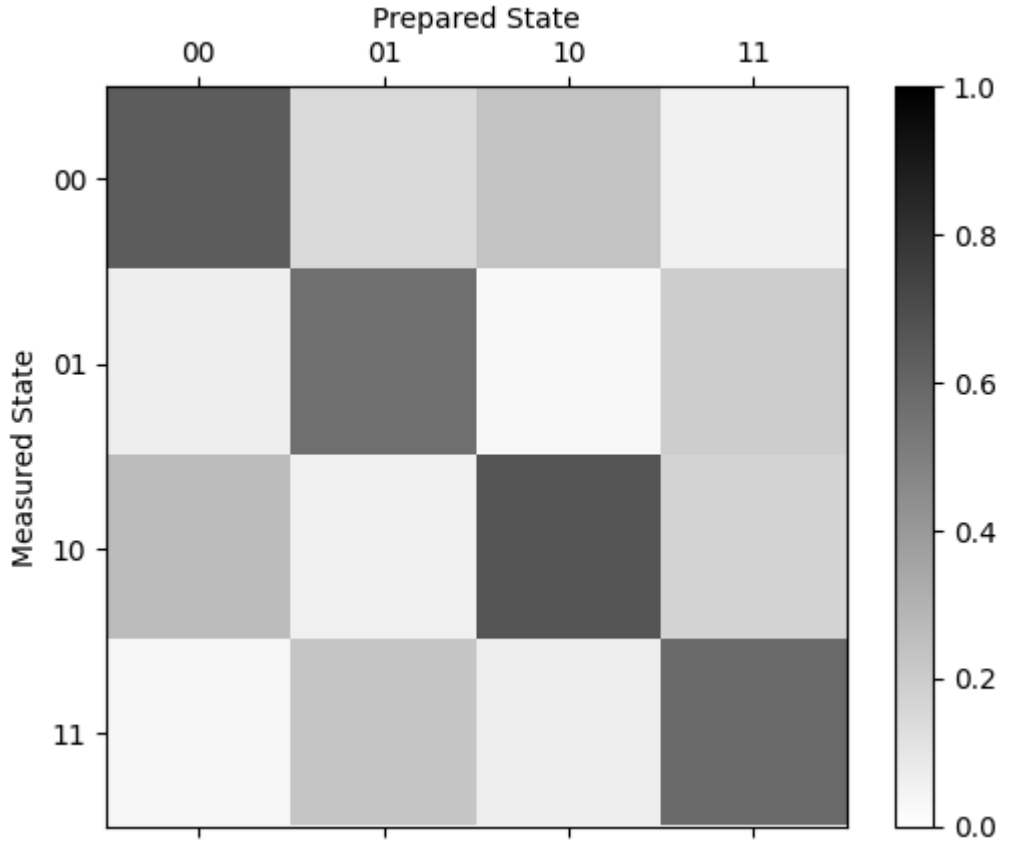


Let's now run these circuits.

[30]:
```
cal_results = [execute_circuit_spin2(qc, qi_backend, shots=2048)[0] for qc in meas_
↪calibs]
```

With the results we can construct the calibration matrix, which we have been calling M.

[31]:
```
meas_fitter = CompleteMeasFitter(cal_results, state_labels, circlabel="")
meas_fitter.cal_matrix

meas_fitter.plot_calibration()
```



[32]:
```
print(f"Readout fidelity of our system: {meas_fitter.readout_fidelity():.2f}")
```

```
Readout fidelity of our system: 0.61
```

Want to know more?

Note that the error mitigation protocol described above does not change the execution of the quantum algorithms. There is another mitigation technique called readout rebalancing that does change the executed algorithms. Readout rebalancing places strategic X gates before measurements in order to reduce the variability in the measurement results. Check out the paper Readout Rebalancing for Near Term Quantum Computers and the corresponding notebook for more information.

Back to the *main page*.

1.3 Python Examples

1.3.1 cQASM examples

Grover's Algorithm: implementation in cQASM and performance analysis

This example demonstrates how to use the SDK to create a more complex Grover's algorithm in cQASM, and simulate the circuit on Quantum Inspire.

```
import os

from quantuminspire.api import QuantumInspireAPI
from quantuminspire.credentials import get_authentication

from src.run import generate_sat_qasm, execute_sat_qasm

QI_URL = os.getenv('API_URL', 'https://api.quantum-inspire.com/')


# Authenticate, create project and define backend
name = 'SAT_Problem'
authentication = get_authentication()
qi = QuantumInspireAPI(QI_URL, authentication, project_name=name)
backend = qi.get_backend_type_by_name(
    'QX single-node simulator')  # if the project already exists in My QI, the existing␣
↪backend will be used and this line will not overwrite the backend information.
shot_count = 512

# define SAT problem
boolean_expr = "(a and not(b) and c and d) or (not(a) and b and not(c) and d)"

# choose variables; see src.run.generate_sat_qasm() for details.
# recommended settings for a simulator
cnot_mode = "normal"
sat_mode = "reuse qubits"
apply_optimization_to_qasm = False
connected_qubit = None

# # recommended settings for Starmon-5. As of now, any sat problem involving more than␣
↪one variable will contain too many gates,
# # and decoherence will occur. If you try this, be sure to use a boolean_expr and sat_
↪mode that initiates a total of 5 qubits.
# cnot_mode = "no toffoli"
# sat_mode = "reuse qubits"
# apply_optimization_to_qasm = False
# connected_qubit = '2'


# generate the qasm code that will solve the SAT problem
qasm, _, qubit_count, data_qubits = generate_sat_qasm(expr_string=boolean_expr, cnot_
↪mode=cnot_mode, sat_mode=sat_mode,
                                                      apply_optimization=apply_
```

(continues on next page)

(continued from previous page)

```
↪optimization_to_qasm,
                                                    connected_qubit=connected_qubit)

print(qasm)

# execute the qasm code and show the results
execute_sat_qasm(qi=qi, qasm=qasm, shot_count=shot_count, backend=backend, qubit_
↪count=qubit_count,
                 data_qubits=data_qubits, plot=True)
```

Code for generating and executing the cQASM

```
from src.optimizer import *
from src.sat_utilities import *
import math


def generate_sat_qasm(expr_string, cnot_mode, sat_mode, apply_optimization=True,␣
↪connected_qubit=None):
    """
    Generate the QASM needed to evaluate the SAT problem for a given boolean expression.

    Args:
        expr_string: A boolean expression as a string
        cnot_mode: The mode for CNOTs. 'normal' and 'no toffoli' are verified to be working.
↪ 'crot' and 'fancy cnot' are experimental.
              - normal: use toffoli gates and ancillary qubits for max speed
              - no toffoli: same as normal, but replace toffoli gates for 2-gate␣
↪equivalent circuits. uses ancillary qubits. This mode must be used if using a backend␣
↪that doesn't support toffoli gates, like starmon-5.
              - crot: no ancillary qubits or toffoli gates, but scales with 3^n gates␣
↪for n bits
              - fancy cnot: no ancillary qubits or toffoli gates, scales 2^n
        sat_mode: The mode for the SAT solving circuit:
              - reuse gates: use minimal amount of gates
              - reuse qubits: use minimal amount of ancillary qubits
        apply_optimization: Whether to apply the optimization algorithm to our generated␣
↪QASM, saving ~20-50% lines of code
        connected_qubit: This functionallity is meant for backends in which not all␣
↪qubits are connected to each other.
        For example, in Starmon-5, only the third qubit is connected to all other qubits.
↪ In order to make the qasm work on
        this backend, connected_qubit='2' should be given as argument. Qubits are then␣
↪swapped during the algorithm such that every gate is
        between the connected_qubit and another one. This function can only be used in␣
↪combination with
        cnot_mode='no toffoli', to ensure no three qubit gates are present.
              - None: do not swap any gates
              - '2': swap qubits to ensure all gates involve qubit 2.
```

(continues on next page)

(continued from previous page)

```
    Returns: A tuple of the following values:
        - qasm: The QASM representing the requested Grover search
        - line count: The total number of parallel lines that is executed (including␣
↪grover loops)
        - qubit count: The total number of qubits required
        - data qubits: The number of data qubits
    """

    algebra = boolean.BooleanAlgebra()
    expr = algebra.parse(expr_string)

    control_names = sorted(list(expr.symbols), reverse=True)

    # note that the number of data qubits also includes an extra bit which must be 1 for␣
↪the algorithm to succeed
    data_qubits = len(control_names) + 1

    expr = split_expression_evenly(expr.simplify())

    if sat_mode == "reuse gates":
        oracle_qasm, _, last_qubit_index = generate_sat_oracle_reuse_gates(expr, control_
↪names, is_toplevel=True,
                                                                           mode=cnot_
↪mode)
    elif sat_mode == "reuse qubits":
        oracle_qasm, _, last_qubit_index = generate_sat_oracle_reuse_qubits(expr,␣
↪control_names, [], is_toplevel=True,
                                                                            mode=cnot_
↪mode)
    else:
        raise ValueError("Invalid SAT mode: {} instead of 'reuse gates' or 'reuse qubits'
↪".format(sat_mode))

    qubit_count = last_qubit_index + 1

    # some modes may require many ancillary qubits for the diffusion operator!
    if cnot_mode in ["normal", "no toffoli"]:
        qubit_count = max(qubit_count, data_qubits * 2 - 3)

    qasm = "version 1.0\n" \
           "qubits {}\n".format(qubit_count)

    # initialisation
    qasm += fill("H", data_qubits)

    # looping grover
    iterations = int(math.pi * math.sqrt(2 ** data_qubits - 1) / 4)
    qasm += ".grover_loop({})\n".format(iterations)

    qasm += oracle_qasm + "\n"

    # diffusion
```

(continues on next page)

(continued from previous page)

```
    qasm += fill("H", data_qubits)
    qasm += fill("X", data_qubits)
    qasm += cnot_pillar(cnot_mode, data_qubits)
    qasm += fill("X", data_qubits)
    qasm += fill("H", data_qubits)

    if apply_optimization:
        qasm = apply_optimizations(qasm, qubit_count, data_qubits)

    if connected_qubit is not None:
        qasm = swap_qubits(qasm=qasm, cnot_mode=cnot_mode, apply_optimization=apply_
↪optimization,
                           connected_qubit=connected_qubit)

    # remove blank lines
    qasm_lines = qasm.split("\n")
    qasm_lines = list(filter(lambda x: x not in ["", "{}", " "], qasm_lines))
    qasm = "\n".join(qasm_lines).replace("\n\n", "\n")

    return qasm + "\n.do_measurement\nmeasure_all", iterations * qasm.count("\n"), qubit_
↪count, data_qubits


def execute_sat_qasm(qi, qasm, shot_count, backend, qubit_count, data_qubits, plot):
    """
    Execute the given QASM code and parse the results as though we are evaluating a SAT␣
↪problem.

    Args:
        qi: An instance of the Quantum Inspire API
        qasm: The qasm program
        shot_count: The number of shots to execute on the circuit
        backend: An instance a QI API backend
        qubit_count: The total number of qubits used in the qasm program
        data_qubits: The number of qubits used by Grover's Algorithm (aka non-ancillary)
        plot: Whether to plot the results of this run

    Returns: A tuple of the following values:
        - histogram_list: a list of pairs, specifying a name and probability, as␣
↪returned from QI
        - likely_solutions: a list of bit strings, all of which seem to solve the given␣
↪formula, according to grover
        - runtime: The execution time on the QI backend
    """

    line_count = qasm.count("\n")
    print("Executing QASM code ({} instructions, {} qubits, {} shots)".format(line_count,
↪ qubit_count, shot_count))
    result = qi.execute_qasm(qasm, backend_type=backend, number_of_shots=shot_count)
    runtime = result["execution_time_in_seconds"]
    print("Ran on simulator in {} seconds".format(str(runtime)[:5]))
```

(continues on next page)

(continued from previous page)

```
    if qubit_count > 15:
        print("No plot because of large qubit count")
        histogram_list = interpret_results(result, qubit_count, data_qubits, False)
    else:
        histogram_list = interpret_results(result, qubit_count, data_qubits, plot)

    likely_solutions = []
    print("Interpreting SAT results:")
    highest_prob = max(map(lambda _: _[1], histogram_list))
    for h in histogram_list:
        # remove all ancillaries and the first data bit
        name, prob = h[0][-data_qubits + 1:], h[1]
        if prob > highest_prob / 2:
            print("{} satisfies the SAT problem".format(name))
            likely_solutions.append(name)

    return histogram_list, likely_solutions, runtime
```

Code for the optimizer

```
from src.sat_utilities import alternative_and

optimizations = {
    "HXH": "Z",
    "HH": "",
    "XX": "",
    "ZX": "Y"
}
largest_opt = 3

forbidden = [
    ".grover_loop",
    "Toffoli",
    "CNOT",
    "CR"
]


def apply_optimizations(qasm, qubit_count, data_qubits):
    """
    Apply 3 types of optimization to the given QASM code:
        Combine groups of gates, such as H-X-H, to faster equivalent gates, Z in this␣
↪case.
        Shift gates to be executed in parallel.
        Clean QASM code itself (q[1,2,3] becomes q[1:3])

    Args:
        qasm: Valid QASM code to optimize
        qubit_count: The total number of qubits
        data_qubits: The number of qubits that are not ancillaries
```

(continues on next page)

(continued from previous page)

```
    Returns: A equivalent piece of QASM with optimizations applied
    """

    # run "speed" mode until QASM does not change
    prev_qasm = ""
    while prev_qasm != qasm:
        prev_qasm = qasm[:]
        qasm = optimize(qasm, qubit_count, data_qubits, mode="speed")

    # run "style" mode until QASM does not change
    prev_qasm = ""
    while prev_qasm != qasm:
        prev_qasm = qasm[:]
        qasm = optimize(qasm, qubit_count, data_qubits, mode="style")

    prev_qasm = ""
    while prev_qasm != qasm:
        prev_qasm = qasm[:]
        qasm = optimize_toffoli(qasm)

    # tidy up "ugly" optimized code
    qasm = clean_code(qasm)

    return qasm


def remove_gate_from_line(local_qasm_line, gate_symbol, qubit_index):
    """
    Removes the application of a specific gate on a specific qubit.

    Args:
        local_qasm_line: The line from which this call should be removed.
        gate_symbol: The symbol representing the gate
        qubit_index: The index of the target qubit

    Returns: The same line of QASM, with the gate removed.
    """

    # if gate applied to single qubit, remove gate call entirely
    single_application = "{} q[{}]".format(gate_symbol, qubit_index)
    if single_application in local_qasm_line:
        # if there is a parallel bar right
        local_qasm_line = local_qasm_line.replace(single_application + " | ", "")
        # else: if there is a parallel bar left
        local_qasm_line = local_qasm_line.replace(" | " + single_application, "")
        # else: if it is the only gate in parallelized brackets
        local_qasm_line = local_qasm_line.replace("{" + single_application + "}", "")
        # else: if it is not parellelized at all
        local_qasm_line = local_qasm_line.replace(single_application, "")

    # else remove just the number
```

(continues on next page)

(continued from previous page)

```
    else:
        local_qasm_line = local_qasm_line.replace(",{},".format(qubit_index), ",")
        local_qasm_line = local_qasm_line.replace("[{},".format(qubit_index), "[")
        local_qasm_line = local_qasm_line.replace(",{}]".format(qubit_index), "]")

    return local_qasm_line


def add_gate_to_line(local_qasm_line, gate_symbol, qubit_index):
    """
    Add in parallel the application of a gate on a qubit.
    Args:
        local_qasm_line: The existing line of QASM to add the gate in.
        gate_symbol: The symbol representing the gate.
        qubit_index: The index of the target qubit.

    Returns: The same line of QASM with the gate added.
    """

    # if another operation is already called on this qubit, we have to put the new gate␣
↪on a new line
    if "[" + str(qubit_index) + "]" in local_qasm_line \
            or "[" + str(qubit_index) + "," in local_qasm_line \
            or "," + str(qubit_index) + "," in local_qasm_line \
            or "," + str(qubit_index) + "]" in local_qasm_line:
        local_qasm_line += "\n{} q[{}]\n".format(gate_symbol, qubit_index)

    # if the line is not empty, we need to consider what's already present
    elif local_qasm_line != "":
        # a bracket indicates this line is parallelized with the { gate | gate | gate }␣
↪syntax
        if "{" in local_qasm_line:
            # remove } from the line and add it back at the end
            local_qasm_line = local_qasm_line.rstrip("}| \n") + \
                              " | " + \
                              "{} q[{}]".format(gate_symbol, qubit_index) + \
                              "}\n"

        # no bracket means we have to add the parallelization syntax ourselves
        else:
            local_qasm_line = "{" + local_qasm_line.rstrip("\n") + \
                              " | " + \
                              "{} q[{}]".format(gate_symbol, qubit_index) + "}\n"

    # else, if the line IS empty, we can just put this gate in directly
    else:
        local_qasm_line = "{} q[{}]\n".format(gate_symbol, qubit_index)
    return local_qasm_line


def remove_toffoli_from_line(local_qasm_line, qubit_1, qubit_2, target_qubit):
    """
```

(continues on next page)

(continued from previous page)

```
    Remove a specific Toffoli gate from a line of qasm.

    Args:
        local_qasm_line: The line of qasm
        qubit_1: The first control qubit of the Toffoli gate
        qubit_2: The second control qubit
        target_qubit: The target qubit

    Returns: The same line of qasm without the Toffoli gate call

    """
    single_application = "Toffoli q[{}],q[{}],q[{}]".format(qubit_1, qubit_2, target_
↪qubit)

    # if there is a parallel bar right
    local_qasm_line = local_qasm_line.replace(single_application + " | ", "")
    # else: if there is a parallel bar left
    local_qasm_line = local_qasm_line.replace(" | " + single_application, "")
    # else: if it is the only gate in parallelized brackets
    local_qasm_line = local_qasm_line.replace("{" + single_application + "}", "")
    # else: if it is not parellelized at all
    local_qasm_line = local_qasm_line.replace(single_application, "")
    return local_qasm_line


def add_toffoli_to_line(local_qasm_line, qubit_1, qubit_2, target_qubit):
    """
    Add a single Toffoli gate application to the given line of qasm.

    Args:
        local_qasm_line: The line of qasm
        qubit_1: The first control qubit
        qubit_2: The second control qubit
        target_qubit: The target qubit

    Returns: The same line of qasm with the Toffoli gate added in parallel
    """

    single_application = "Toffoli q[{}],q[{}],q[{}]".format(qubit_1, qubit_2, target_
↪qubit)

    # if the line is not empty, we need to consider what's already present
    if local_qasm_line != "":
        # a bracket indicates this line is parallelized with the { gate_1 | gate_2 |␣
↪gate_3 } syntax
        if "{" in local_qasm_line:
            # remove } from the line and add it back at the end
            local_qasm_line = local_qasm_line.rstrip("}| \n") + \
                              " | " + \
                              single_application + \
                              "}\n"
```

(continues on next page)

(continued from previous page)

```
        # no bracket means we have to add the parallelization syntax ourselves
        else:
            local_qasm_line = "{" + local_qasm_line.rstrip("\n") + \
                              " | " + \
                              single_application + "}\n"

    # else, if the line IS empty, we can just put this gate in directly
    else:
        local_qasm_line = single_application + "\n"
    return local_qasm_line


def optimize(qasm, qubit_count, data_qubits, mode="speed"):
    """
    Apply a single pass of performance-oriented optimizations to the given QASM.

    Args:
        qasm: A valid QASM program to optimize.
        qubit_count: The total number of qubits
        data_qubits: The number of qubits that are not ancillaries
        mode: Setting that determines the type of optimization:
            "speed" -> combine gates into equivalent smaller gates
            "style" -> parallelize gates for speedup and aesthetics

    Returns: Functionally the same QASM code, with one run of optimizations applied.
    """

    qasm_lines = qasm.split("\n")
    gates_applied = []
    for i in range(qubit_count):
        gates_applied.append([])

    for qasm_line_index in range(len(qasm_lines)):
        line = qasm_lines[qasm_line_index]
        if len(line) == 0:
            continue

        gate = line.split()[0].lstrip("{")
        if gate not in ["H", "X", "Y", "Z"]:
            gate = "_"

        if ":" in line:
            raise ValueError("Optimizer does not work well with q[0:3] notation!\n"
                             "Line: {}".format(line))

        if "[" in line:
            for element in line.split(" | "):
                gate = element.split()[0].lstrip("{")
                if gate not in ["H", "X", "Y", "Z"]:
                    gate = "_"

                alt_affected_qubits = []
```

(continues on next page)

(continued from previous page)

```
                for possibly_affected in range(qubit_count):
                    hits = [
                        ",{},".format(possibly_affected),
                        "[{},".format(possibly_affected),
                        ",{}]".format(possibly_affected),
                        "[{}]".format(possibly_affected)
                    ]
                    if any(h in element for h in hits):
                        alt_affected_qubits.append(possibly_affected)

                for a in alt_affected_qubits:
                    gates_applied[a].append((qasm_line_index, gate))
            else:
                for a in range(data_qubits):
                    gates_applied[a].append((qasm_line_index, gate))

    for qubit_index in range(len(gates_applied)):
        gates = gates_applied[qubit_index]
        skip_counter = 0
        for gate_index in range(len(gates)):
            if skip_counter > 0:
                skip_counter -= 1
                continue

            if mode == "speed":
                for offset in range(0, largest_opt - 1):
                    next_gates = "".join(map(lambda _: _[1], gates[gate_index:gate_index␣
↪+ largest_opt - offset]))
                    if next_gates in optimizations:
                        replacement = optimizations[next_gates]

                        line_indices = list(map(lambda _: _[0], gates[gate_index:gate_
↪index + largest_opt - offset]))
                        # first, remove all gates that are to be replaced
                        for idx, line_number in enumerate(line_indices):
                            qasm_lines[line_number] = remove_gate_from_line(qasm_
↪lines[line_number],
                                                                            next_
↪gates[idx],
                                                                            qubit_index)

                        # add replacement gate to first line index
                        # unless there is no replacement gate, of course
                        if replacement != "":
                            qasm_lines[line_indices[0]] = add_gate_to_line(qasm_
↪lines[line_indices[0]],
                                                                           replacement,
                                                                           qubit_index)

                        # ensure we skip a few gates
                        skip_counter += len(next_gates)
```

(continues on next page)

(continued from previous page)

```
            elif mode == "style":
                # check if we can shift left to align with other gates
                current_line, current_gate = gates[gate_index]
                prev_line = current_line - 1

                if any(f in qasm_lines[current_line] or f in qasm_lines[prev_line] for f␣
↪in forbidden):
                    continue
                # if this or the previous line has a break statement, no shifting␣
↪possible
                if current_gate == "_" or (gates[gate_index - 1][1] == "_" and␣
↪gates[gate_index - 1][0] == prev_line):
                    continue

                if qasm_lines[prev_line] == "":
                    continue

                # having passed these checks, we can try to actually shift
                if current_gate in ["H", "X", "Y", "Z"] and str(qubit_index) not in qasm_
↪lines[prev_line]:
                    # remove from current line
                    qasm_lines[current_line] = remove_gate_from_line(qasm_lines[current_
↪line], current_gate,
                                                                     qubit_index)
                    # add to left
                    qasm_lines[prev_line] = add_gate_to_line(qasm_lines[prev_line],␣
↪current_gate, qubit_index)

    # remove blank lines
    qasm_lines = list(filter(lambda x: x not in ["", "{}", " "], qasm_lines))
    return "\n".join(qasm_lines).replace("\n\n", "\n")


def optimize_toffoli(qasm):
    """
    Specific style optimizer capable of left-shifting and annihilating Toffoli gates.

    Args:
        qasm: QASM to optimize

    Returns: Equivalent QASM with Toffoli gates optimized.
    """

    qasm_lines = qasm.split("\n")
    for current_line_index in range(1, len(qasm_lines)):
        cur_line = qasm_lines[current_line_index]
        prev_line = qasm_lines[current_line_index - 1]
        if "Toffoli" in cur_line and "Toffoli" in prev_line and ".grover_loop" not in␣
↪prev_line:
            # find all Toffoli triplets in both lines
            prev_line_gates = prev_line.strip("{} |").split(" | ")
            prev_line_toffolis = list(filter(lambda x: "Toffoli" in x, prev_line_gates))
```

(continues on next page)

(continued from previous page)

```
            cur_line_gates = cur_line.strip("{} |").split(" | ")
            cur_line_toffolis = list(filter(lambda x: "Toffoli" in x, cur_line_gates))

            any_updated = False
            for c_t in cur_line_toffolis:
                if any_updated:
                    break

                c_qubit_1, c_qubit_2, c_target_qubit = tuple(map(int, c_t.strip("Toffoli␣
↪q[]").split("],q[")))
                shiftable = True

                for p_t in prev_line_toffolis:
                    p_qubit_1, p_qubit_2, p_target_qubit = tuple(map(int, p_t.strip(
↪"Toffoli q[]").split("],q[")))

                    if {c_qubit_1, c_qubit_2} == {p_qubit_1, p_qubit_2} and c_target_
↪qubit == p_target_qubit:
                        # remove toffolis from both lines
                        cur_line = remove_toffoli_from_line(cur_line, c_qubit_1, c_qubit_
↪2, c_target_qubit)
                        prev_line = remove_toffoli_from_line(prev_line, p_qubit_1, p_
↪qubit_2, p_target_qubit)
                        shiftable = False
                        any_updated = True
                        break
                    elif len({c_qubit_1, c_qubit_2, c_target_qubit}.intersection(
                            {p_qubit_1, p_qubit_2, p_target_qubit})) != 0:
                        shiftable = False
                        break

                # check if anything has blocked us from shifting left
                if shiftable:
                    # otherwise we can go!
                    cur_line = remove_toffoli_from_line(cur_line, c_qubit_1, c_qubit_2,␣
↪c_target_qubit)
                    prev_line = add_toffoli_to_line(prev_line, c_qubit_1, c_qubit_2, c_
↪target_qubit)

                    any_updated = True

            if any_updated:
                qasm_lines[current_line_index] = cur_line
                qasm_lines[current_line_index - 1] = prev_line

    # remove blank lines
    qasm_lines = list(filter(lambda x: x not in ["", "{}", " "], qasm_lines))

    return "\n".join(qasm_lines).replace("\n\n", "\n")
```

(continues on next page)

(continued from previous page)

```
def replace_toffoli_with_alt(qasm):
    """
    Replace all Toffoli gates (including parallelized ones) by their alternative␣
↪representation.
    See src.grover.search_utilies.alternative_toffoli for more details.

    Args:
        qasm: The full qasm program that contains Toffoli gates to replace

    Returns: The same qasm with Toffoli gates replaced.
    """
    qasm_lines = qasm.split("\n")
    for current_line_index in range(len(qasm_lines)):
        cur_line = qasm_lines[current_line_index]
        if "Toffoli" in cur_line:
            # find all Toffoli triplets in this line

            cur_line_gates = cur_line.strip("{} |").split(" | ")
            cur_line_toffolis = list(filter(lambda x: "Toffoli" in x, cur_line_gates))

            multiple = len(cur_line_toffolis) > 1

            line_strings = []
            for i in range(7):
                if multiple:
                    line_strings.append("{")
                else:
                    line_strings.append("")

            for current_t in cur_line_toffolis:
                qubit_1, qubit_2, target_qubit = tuple(map(int, current_t.strip("Toffoli␣
↪q[]").split("],q[")))
                alt_qasm_lines = alternative_and(qubit_1, qubit_2, target_qubit).split("\
↪n")
                for j in range(7):
                    line_strings[j] += alt_qasm_lines[j]
                    if multiple:
                        line_strings[j] += " | "

            if multiple:
                for i in range(7):
                    line_strings[i] = line_strings[i][:-3] + "}"

            cur_line = "\n".join(line_strings)

            qasm_lines[current_line_index] = cur_line

    # remove blank lines
    qasm_lines = list(filter(lambda x: x not in ["", "{}", " "], qasm_lines))
    return "\n".join(qasm_lines).replace("\n\n", "\n")
```

(continues on next page)

(continued from previous page)

```
def clean_code(qasm):
    """
    Clean given QASM by rewriting each line to a more readable format.
    For example, "{ X q[0] | H q[3,4,5] | X q[1] | X q[2] }"
    Would become "{ X q[0:2] | H q[3:5]"

    Args:
        qasm: Valid QASM code to clean

    Returns: The same QASM code with improved formatting.
    """

    qasm_lines = qasm.split("\n")
    for idx in range(len(qasm_lines)):
        line = qasm_lines[idx]
        gate_dict = {}
        new_line = ""
        if "Toffoli" not in line and "CR" not in line and "CNOT" not in line and ("{" in␣
↪line or "," in line):
            line = line.strip("{}")
            elements = line.split("|")
            for e in elements:
                gate, target = e.split()
                indices = list(map(int, target.strip("q[]").split(",")))
                if gate not in gate_dict:
                    gate_dict[gate] = indices
                else:
                    gate_dict[gate] += indices

            parallel = len(gate_dict.keys()) > 1
            if parallel:
                new_line += "{ "
            for gate, indices in gate_dict.items():
                if max(indices) - min(indices) + 1 == len(indices) > 1:
                    new_line += "{} q[{}:{}]".format(gate, min(indices), max(indices))
                else:
                    new_line += "{} q[{}]".format(gate, ",".join(map(str, indices)))
                new_line += " | "

            new_line = new_line[:-3]
            if parallel:
                new_line += " }"
        else:
            new_line = line

        qasm_lines[idx] = new_line

    return "\n".join(qasm_lines)
```

Code for the SAT-utilities

```
import boolean
from boolean.boolean import AND, OR, NOT, Symbol
import random

import matplotlib.pyplot as plt
import math


def apply(gate, qubit):
    """
    Simply apply a gate to a single qubit

    Args:
        gate: The gate to apply
        qubit: The target qubit

    Returns: Valid QASM that represents this application

    """
    return "{} q[{}]\n".format(gate, qubit)


def fill(character, data_qubits):
    """
    Apply a specific gate to all data qubits
    Args:
        character: The QASM gate to apply
        data_qubits: The number of data qubits

    Returns: Valid QASM to append to the program
    """
    # create a list of the qubit indices that need the gate applied
    indices = ",".join(map(str, range(data_qubits)))

    return "{} q[{}]\n".format(character, indices)


def normal_n_size_cnot(n, mode):
    """
    Generate a CNOT with n control bits.
    It is assumed the control bits have indices [0:n-1],
    and the target bit is at index [n].

    Args:
        n: The number of control bits
        mode: The method by which we will make CNOT gates

    Returns: Valid QASM to append to the program
    """
```

(continues on next page)

(continued from previous page)

```
    if n == 1:
        local_qasm = "CNOT q[0],q[1]\n"
    elif n == 2:
        if mode == "no toffoli":
            local_qasm = alternative_and(0, 1, 2)
        else:
            local_qasm = "Toffoli q[0],q[1],q[2]\n"
    else:
        # for n > 2, there is no direct instruction in QASM, so we must generate an␣
↪equivalent circuit
        # the core idea of a large CNOT is that we must AND-gate together all the␣
↪control bits
        # we do this with Toffoli gates, and store the result of each Toffoli on␣
↪ancillary qubits

        # we keep a list of all the bits that should be AND-ed together
        bits_to_and = list(range(n))
        ancillary_count = 0

        # make a list of all the Toffoli gates to eventually write to the program
        gate_list = []

        # we will continue looping until all bits are and-ed together
        while len(bits_to_and) > 0:
            # take the first two
            a, b = bits_to_and[:2]
            # if these are the only two elements to AND, we're almost done!
            # just combine these 2 in a Toffoli...
            # ...which targets the global "target bit" of this n-CNOT
            if len(bits_to_and) == 2:
                target = n
                bits_to_and = []

            # the default case is to write the result to an ancillary bit
            else:
                target = n + 1 + ancillary_count
                ancillary_count += 1
                # remove the used qubits from the list of bits to AND
                bits_to_and = bits_to_and[2:] + [target]

            if mode == "no toffoli":
                gate_list.append(alternative_and(a, b, target))
            else:
                gate_list.append("Toffoli q[{}],q[{}],q[{}]".format(a, b, target))
            # gate_list.append("Toffoli q[{}],q[{}],q[{}]".format(a, b, target))

        # Apply the complete list of gates in reverse after the target is flipped
        # This undoes all operations on the ancillary qubits (so they remain 0)
        gate_list = gate_list + gate_list[-2::-1]
        local_qasm = "\n".join(gate_list) + "\n"

    return local_qasm
```

(continues on next page)

(continued from previous page)

```
def n_size_crot(n, start_n, target, angle):
    """
    Generate a controlled rotation with n control bits without using Toffoli gates.
    It is assumed the control bits have indices [start_n : start_n + n-1].

    Args:
        n: The number of control bits
        start_n: The first index that is a control bit
        target: The target bit index
        angle: The angle in radians by which to shift the phase
               An angle of pi gives an H-Z-H aka X gate
               An angle of pi/2 gives an H-S-H gate
               An angle of pi/4 gives an H-T-H gate
               Etc.

    Returns: Valid QASM to append to the program
    """
    local_qasm = ""

    if n == 1:
        # Simply a CROT with the given angle
        local_qasm += apply("H", target)
        local_qasm += "CR q[{}],q[{}],{}\n".format(start_n, target, angle)
        local_qasm += apply("H", target)
    else:
        # V gate using the lowest control bit
        local_qasm += n_size_crot(1, start_n + n - 1, target, angle / 2)

        # n-1 CNOT on highest bits (new_angle = angle)
        local_qasm += n_size_crot(n - 1, 0, start_n + n - 1, math.pi)

        # V dagger gate on lowest two bits
        local_qasm += n_size_crot(1, start_n + n - 1, target, -angle / 2)

        # n-1 CNOT on highest bits (new_angle = angle)
        local_qasm += n_size_crot(n - 1, 0, start_n + n - 1, math.pi)

        # controlled V gate using highest as controls and lowest as target (new_angle =␣
↪angle / 2)
        local_qasm += n_size_crot(n - 1, 0, target, angle / 2)

    return local_qasm


def cnot_pillar(mode, data_qubits):
    """
    Generate a common structure that applies a Hadamard, CNOT, and Hadamard again to the␣
↪lowest data bit

    Returns: Valid QASM to append to the program
```

(continues on next page)

(continued from previous page)

```
    """
    local_qasm = "H q[{}]\n".format(data_qubits - 1)
    if mode in ["normal", "no toffoli"]:
        local_qasm += normal_n_size_cnot(data_qubits - 1, mode)
    elif mode == "crot":
        local_qasm += n_size_crot(data_qubits - 1, 0, data_qubits - 1, math.pi)
    elif mode == "fancy cnot":
        local_qasm += fancy_cnot(data_qubits - 1)
    local_qasm += "H q[{}]\n".format(data_qubits - 1)
    return local_qasm


def search_oracle(search_term, data_qubits):
    """
    Generate a common structure that is used in the oracle circuit.
    It flips all bits that correspond to a 0 in the search target.
    Args:
        search_term: The search term for which to generate an oracle
        data_qubits: The number of data qubits

    Returns: Valid QASM to append to the program
    """
    if "0" not in search_term:
        return "\n"

    local_qasm = "X q["
    for i in range(data_qubits):
        if search_term[i] == "0":
            local_qasm += str(i) + ","

    # remove last comma and add closing bracket
    local_qasm = local_qasm[:-1] + "]\n"
    return local_qasm


def int_to_bits(int_str, qubit_count):
    """
    Convert a number (possibly in string form) to a readable bit format.
    For example, the result '11', which means both qubits were measured as 1, is returned␣
↪by the API as "3".
    This converts that output to the readable version.

    Args:
        int_str: A string or integer in base 10 that represents the measurement outcome.

    Returns: A string of 0's and 1's
    """
    # convert to an integer, then generate the binary string
    # remove the "0b" prefix from the binary string
    # then pad (using zfill) with 0's
    return str(bin(int(int_str)))[2:].zfill(qubit_count)
```

(continues on next page)

(continued from previous page)

```
def interpret_results(result_dict, qubit_count, data_qubits, plot=True):
    """
    Parse the result dictionary given by the API into a readable format, and plot it.

    Args:
        result_dict: The dictionary given by qi.execute_qasm
        plot: Whether to plot the results in a bar chart

    Returns: Parsed result
    """

    num_of_measurements = 2 ** qubit_count

    # we still store the histogram bars in here, and later sort them in ascending order
    ordered_bars = [None for _ in range(num_of_measurements)]

    for i in range(num_of_measurements):
        # find result in dictionary and add to list of bars
        # zero-valued bars don't show up in the dictionary, hence the try-except
        try:
            bar = result_dict["histogram"][str(i)]
        except KeyError:
            bar = 0

        # generate corresponding binary name (so "11" instead of "3")
        name = int_to_bits(i, qubit_count)

        ordered_bars[i] = (name, bar)

    if plot:
        for b in ordered_bars:
            # check if the given bar has 0's for all the ancillary qubits
            # if it does not, we assume it is irrelevant for the histogram, so we don't␣
↪plot it
            if int(b[0], 2) < 2 ** data_qubits:
                plt.bar(b[0][-data_qubits:], b[1])
            # if a result is returned where some ancillary qubits were not zero, we have␣
↪a problem
            elif b[1] != 0:
                raise ValueError("\tNonzero result from 'impossible' measurement:\n"
                                 "\tColumn {} has fraction {}. This means not all␣
↪control bits were 0!".format(b[0],
                                                                                    ␣
↪                      b[1]))

        # set styling for the x-axis markers
        plt.xticks(fontsize=6, rotation=45, ha="right")
        plt.title("Measurements, discarding ancilla qubits")
        plt.show()

    return ordered_bars
```

(continues on next page)

(continued from previous page)

```
def gray_code(n):
    """
    Generate a Gray code sequence of bit string with length n.

    Args:
        n: The size for each element in the Gray code

    Returns: An array of strings forming a Gray code
    """

    if n == 1:
        return ["0", "1"]
    else:
        g_previous = gray_code(n - 1)
        mirrored_paste = g_previous + g_previous[::-1]
        g_current = mirrored_paste[:]
        for i in range(2 ** n):
            if i < 2 ** (n - 1):
                g_current[i] = g_current[i] + "0"
            else:
                g_current[i] = g_current[i] + "1"
        return g_current


def fancy_cnot(n):
    """
    Generate a circuit equivalent to an n-bit CNOT.
    This avoids using Toffoli gates or ancillary qubits.
    Args:
        n: Number of control bits

    Returns: Valid QASM that represents a CNOT

    """
    gray_code_list = gray_code(n)[1:]
    local_qasm = apply("H", n)

    for i in range(len(gray_code_list)):
        if i == 0:
            local_qasm += "CR q[0],q[{}],{}\n".format(n, math.pi / (2 ** (n - 1)))
        else:
            prev_gray = gray_code_list[i - 1]
            cur_gray = gray_code_list[i]

            flip_idx = -1
            for j in range(len(cur_gray)):
                if cur_gray[j] != prev_gray[j]:
                    flip_idx = j
                    break
```

(continues on next page)

(continued from previous page)

```
            last_1_bit_cur = len(cur_gray) - 1 - cur_gray[::-1].index("1")
            last_1_bit_prev = len(prev_gray) - 1 - prev_gray[::-1].index("1")

            bit_a = flip_idx

            if flip_idx == last_1_bit_cur:
                bit_b = last_1_bit_prev
            else:
                bit_b = last_1_bit_cur

            control_bit = min(bit_a, bit_b)
            target_bit = max(bit_a, bit_b)

            local_qasm += "CNOT q[{}],q[{}]\n".format(control_bit, target_bit)

            parity = cur_gray.count("1") % 2
            if parity == 0:
                angle = -math.pi / (2 ** (n - 1))
            else:
                angle = math.pi / (2 ** (n - 1))

            local_qasm += "CR q[{}],q[{}],{}\n".format(target_bit, n, angle)

    local_qasm += apply("H", n)
    return local_qasm


def alternative_and(control_1, control_2, target):
    """
    Generate a circuit from 1 and 2 qubit gates that performs an operation equivalent to␣
↪a Toffoli gate.

    Args:
        control_1: First control bit index
        control_2: Second control bit index
        target: Target bit index

    Returns: Valid QASM that performs a CCNOT
    """

    if control_1 == control_2:
        return "CNOT q[{}],q[{}]\n".format(control_1, target)

    local_qasm = ""
    local_qasm += apply("H", target)
    local_qasm += f"CNOT q[{control_2}],q[{target}]\n"
    local_qasm += apply("Tdag", target)
    local_qasm += f"CNOT q[{control_1}],q[{target}]\n"
    local_qasm += apply("T", target)
    local_qasm += f"CNOT q[{control_2}],q[{target}]\n"
    local_qasm += apply("Tdag", target)
    local_qasm += f"CNOT q[{control_1}],q[{target}]\n"
```

(continues on next page)

(continued from previous page)

```
    local_qasm += apply("T", control_2)
    local_qasm += apply("T", target)
    local_qasm += apply("H", target)
    local_qasm += f"CNOT q[{control_1}],q[{control_2}]\n"
    local_qasm += apply("T", control_1)
    local_qasm += apply("Tdag", control_2)
    local_qasm += f"CNOT q[{control_1}],q[{control_2}]\n"

    return local_qasm


def generate_sat_oracle_reuse_gates(expr: boolean.Expression, control_names, is_
↪toplevel=False, mode='normal'):
    """
    Generate the circuit needed for an oracle solving the SAT problem, given a boolean␣
↪expression.
    This uses a new ancillary qubit for every boolean gate, UNLESS that sub-expression␣
↪has already been calculated before.

    Args:
        expr: The boolean expression (instance of boolean.Expression, not a string)
        control_names: The names of the control variables
        is_toplevel: Whether this is the main call, or a recursive call

    Returns: A tuple of the following values:
        - qasm: The QASM for this expression
        - target_qubit: The qubit line on which the output of this expression is placed
    """

    global highest_qubit_used
    global expressions_calculated

    if is_toplevel:
        highest_qubit_used = len(control_names)
        expressions_calculated = {}
    local_qasm = ""

    # go through possible types
    if type(expr) == AND or type(expr) == OR:
        # left side
        left_qasm, left_qubit, _ = generate_sat_oracle_reuse_gates(expr.args[0], control_
↪names, mode=mode)
        expressions_calculated[expr.args[0]] = left_qubit
        right_qasm, right_qubit, _ = generate_sat_oracle_reuse_gates(expr.args[1],␣
↪control_names, mode=mode)
        expressions_calculated[expr.args[1]] = right_qubit
        local_qasm += left_qasm
        local_qasm += right_qasm
    elif type(expr) == NOT:
        inner_qasm, inner_qubit, _ = generate_sat_oracle_reuse_gates(expr.args[0],␣
↪control_names, mode=mode)
        local_qasm += inner_qasm
```

(continues on next page)

(continued from previous page)

```
        local_qasm += "X q[{}]\n".format(inner_qubit)
        return local_qasm, inner_qubit, highest_qubit_used
    elif type(expr) == Symbol:
        # nothing to do here
        return local_qasm, control_names.index(expr), highest_qubit_used
    else:
        raise ValueError("Unknown boolean expr type: {}".format(type(expr)))

    if expr in expressions_calculated:
        already_calculated_index = expressions_calculated[expr]
        # we don't need to add any qasm, just say where this expression can be found
        return "", already_calculated_index, highest_qubit_used

    if is_toplevel:
        target_qubit = len(control_names)
        local_qasm += "H q[{}]\n".format(len(control_names))
        left_half_qasm = local_qasm[:]
    else:
        # we need another ancillary bit
        highest_qubit_used += 1
        target_qubit = highest_qubit_used

    if type(expr) == AND:
        if mode == 'normal':
            local_qasm += generate_and(left_qubit, right_qubit, target_qubit)
        else:
            local_qasm += alternative_and(left_qubit, right_qubit, target_qubit)
    elif type(expr) == OR:
        if mode == 'normal':
            local_qasm += generate_or(left_qubit, right_qubit, target_qubit)
        else:
            local_qasm += alternative_or(left_qubit, right_qubit, target_qubit)

    # undo NOT applications
    if len(expr.args) == 2 and not is_toplevel:
        if type(expr.args[0]) == NOT:
            local_qasm += "X q[{}]\n".format(left_qubit)
        if type(expr.args[1]) == NOT:
            local_qasm += "X q[{}]\n".format(right_qubit)

    # indicate to other calls of this function that this expression has been generated␣
↪already
    expressions_calculated[expr] = target_qubit

    if is_toplevel:
        local_qasm += "\n".join(left_half_qasm.split("\n")[::-1])
        return local_qasm, target_qubit, highest_qubit_used

    return local_qasm, target_qubit, highest_qubit_used


def generate_sat_oracle_reuse_qubits(expr, control_names, avoid, last_qubit=-1, is_
```

(continues on next page)

(continued from previous page)

```
↪toplevel=False, mode='normal'):
    """
    Generate a SAT oracle that saves on ancillary qubits by resetting them, so that they␣
↪can be reused.

    Args:
        expr: The boolean expression to generate an oracle for
        avoid: The ancillary lines that we can't use because they already contain data␣
↪(default is an empty list)
        control_names: The names of the variables in the expression (such as "a", "b"␣
↪and "c")
        last_qubit: The highest qubit index we have ever used. This is needed to␣
↪calculate the total number of ancillaries
        is_toplevel: Whether this function call is the "original". In that case, its␣
↪output is a specific qubit line

    Returns: A tuple of the following values:
        - target_line: The output line of the qasm representing the input expression
                       All other lines are guaranteed to be reset to 0
        - qasm: The QASM code
        - last_qubit: The highest ancillary qubit index encountered in this expression
    """

    first_ancillary_bit = len(control_names) + 1

    if len(expr.args) > 2:
        raise ValueError("Fancy SAT Oracle expects only 1 and 2-argument expressions,␣
↪but got {}".format(expr.args))

    if type(expr) == Symbol:
        return "", control_names.index(expr), last_qubit
    elif type(expr) == NOT:
        qubit_index = control_names.index(expr.args[0])
        return "X q[{}]".format(qubit_index), qubit_index, last_qubit
    elif type(expr) == AND:
        if mode == 'normal':
            generate_func = generate_and
        else:
            generate_func = alternative_and
    elif type(expr) == OR:
        if mode == 'normal':
            generate_func = generate_or
        else:
            generate_func = alternative_or
    else:
        raise ValueError("Unknown type in Boolean expression: {}".format(type(expr)))

    left_expr = expr.args[0]
    right_expr = expr.args[1]

    left_qasm, left_target_qubit, left_last_qubit = generate_sat_oracle_reuse_
↪qubits(left_expr, control_names,
```

(continues on next page)

(continued from previous page)

```
↪avoid[:],
↪last_qubit, mode=mode)
    avoid.append(left_target_qubit)
    right_qasm, right_target_qubit, right_last_qubit = generate_sat_oracle_reuse_
↪qubits(right_expr, control_names,
↪avoid[:],
↪last_qubit, mode=mode)
    avoid.append(right_target_qubit)

    target_qubit = -1
    # if toplevel, we know what to target: the specific line that is set to the |1> state
    if is_toplevel:
        target_qubit = first_ancillary_bit - 1
        my_qasm = "H q[{}]\n".format(target_qubit) + \
                  generate_func(left_target_qubit, right_target_qubit, target_qubit) + \
                  "H q[{}]\n".format(target_qubit)
    else:
        # find the lowest line we can use
        # if necessary, we can target an entirely new line (if all the others are used)
        for i in range(first_ancillary_bit, first_ancillary_bit + max(avoid) + 1):
            if i not in avoid:
                target_qubit = i
                break
        my_qasm = generate_func(left_target_qubit, right_target_qubit, target_qubit)

    last_qubit = max(last_qubit, max(avoid), left_last_qubit, right_last_qubit, target_
↪qubit)

    local_qasm = "\n".join([
        left_qasm,
        right_qasm,
        my_qasm,
        *right_qasm.split("\n")[::-1],
        *left_qasm.split("\n")[::-1]
    ])

    return local_qasm, target_qubit, last_qubit


def generate_and(qubit_1, qubit_2, target_qubit):
    """
    Generate an AND in qasm code (just a Toffoli).
    """
    if qubit_1 == qubit_2:
        return "CNOT q[{}],q[{}]\n".format(qubit_1, target_qubit)

    return "Toffoli q[{}],q[{}],q[{}]\n".format(qubit_1, qubit_2, target_qubit)
```

(continues on next page)

(continued from previous page)

```
def generate_or(qubit_1, qubit_2, target_qubit):
    """
    Generate an OR in qasm code (Toffoli with X gates).
    """
    if qubit_1 == qubit_2:
        return "CNOT q[{}],q[{}]\n".format(qubit_1, target_qubit)

    local_qasm = "X q[{},{}]\n".format(qubit_1, qubit_2)
    local_qasm += generate_and(qubit_1, qubit_2, target_qubit)
    local_qasm += "X q[{},{},{}]\n".format(qubit_1, qubit_2, target_qubit)
    return local_qasm


def alternative_or(qubit_1, qubit_2, target_qubit):
    if qubit_1 == qubit_2:
        return "CNOT q[{}],q[{}]\n".format(qubit_1, target_qubit)

    local_qasm = "X q[{},{}]\n".format(qubit_1, qubit_2)
    local_qasm += alternative_and(qubit_1, qubit_2, target_qubit)
    local_qasm += "X q[{},{},{}]\n".format(qubit_1, qubit_2, target_qubit)
    return local_qasm


def split_expression_evenly(expr):
    """
    Split a Boolean expression as evenly as possible into a binary tree.

    Args:
        expr: The Boolean expression to split

    Returns: The same expression, where all gates are applied to exactly 2 elements
    """

    expr_type = type(expr)
    if len(expr.args) > 2:
        halfway = int(len(expr.args) / 2)
        right_expanded = split_expression_evenly(expr_type(*expr.args[halfway:]))

        if len(expr.args) > 3:
            left_expanded = split_expression_evenly(expr_type(*expr.args[:halfway]))
        else:
            left_expanded = split_expression_evenly(expr.args[0])

        return expr_type(left_expanded, right_expanded)
    elif len(expr.args) == 2:
        return expr_type(split_expression_evenly(expr.args[0]),
                         split_expression_evenly(expr.args[1]))
    else:
        return expr
```

(continues on next page)

(continued from previous page)

```
def generate_ksat_expression(n, m, k):
    """
    Generate an arbitrary k-SAT expression according to the given parameters.

    Args:
        n: The number of groups
        m: The number of variables in a group
        k: The number of variables

    Returns: A Boolean expression
    """
    if m > k:
        raise ValueError("m > k not possible for kSAT")

    alphabet = []
    for i in range(k):
        alphabet.append(chr(97 + i))

    expression = ""

    for i in range(n):
        literals = random.sample(alphabet, m)
        expression += " and ({}".format(literals[0])
        for l in literals[1:]:
            if random.random() < 0.5:
                expression += " or not({})".format(l)
            else:
                expression += " or {}".format(l)
        expression += ")"

    return expression.lstrip("and ")


def swap_qubits(qasm, cnot_mode, apply_optimization, connected_qubit='2'):
    """
    Implement swap gates to ensure all gates are between the connected_qubit and some␣
↪other qubit. This is necesarry in
    for example starmon-5 QPU backend, as only qubit 2 is connected to all other qubits.
    For example, "CNOT q[1],q[4]"
    Would become "SWAP q[1],q[2]
                  CNOT q[2],q[4]
                  SWAP q[1],q[2]"   if connected_qubit='2'

    Args:
        qasm: Valid QASM code
        connected_qubit: The qubit that is connected to all other qubits. For starmon-5␣
↪this is the third qubit, so connected_qubit='2'.

    Returns: functionally equal QASM code that only has gates between connected qubits.

    This function should not be used on optimized code, or on code that contains three␣
↪qubit gates. In practice, this
```

(continues on next page)

(continued from previous page)

```
    means you should only apply the function when using the mode cnot_mode='no toffoli'␣
↪and apply_optimization=False
    """
    if connected_qubit is None:
        return qasm

    if cnot_mode != 'no toffoli' or apply_optimization:
        raise ValueError(
            "Invalid mode: can only use swap_qubits() in combination with cnot_mode='no␣
↪toffoli' and apply_optimization=False")

    new_lines = []
    for line in qasm.split('\n'):
        if ' ' in line:
            args = line.split(' ')[1].split(',')
            if len(args) > 1:
                if args[0][0] == 'q' and args[1][0] == 'q':
                    q0 = args[0][2]
                    q1 = args[1][2]
                    if q0 != connected_qubit and q1 != connected_qubit:
                        to_swap = min([q0, q1])
                        swap_arg = f'SWAP q[{to_swap}],q[{connected_qubit}]'
                        new_lines += [swap_arg, line.replace(to_swap, connected_qubit),␣
↪swap_arg]
                        continue
        new_lines += [line]
    return '\n'.join(new_lines)
```

1.3.2 ProjectQ examples

ProjectQ example 1

A simple example that demonstrates how to use the SDK to create a circuit to create a Bell state, and simulate the circuit on Quantum Inspire.

```
"""
This example is copied from https://github.com/ProjectQ-Framework/ProjectQ
and is covered under the Apache 2.0 license.
"""
import os

from projectq import MainEngine
from projectq.backends import ResourceCounter
from projectq.ops import CNOT, H, Measure, All
from projectq.setups import restrictedgateset

from quantuminspire.api import QuantumInspireAPI
from quantuminspire.credentials import get_authentication
from quantuminspire.projectq.backend_qx import QIBackend

QI_URL = os.getenv('API_URL', 'https://api.quantum-inspire.com/')
```

(continues on next page)

(continued from previous page)

```
project_name = 'ProjectQ-entangle'
authentication = get_authentication()
qi_api = QuantumInspireAPI(QI_URL, authentication, project_name=project_name)
qi_backend = QIBackend(quantum_inspire_api=qi_api)

compiler_engines = restrictedgateset.get_engine_list(one_qubit_gates=qi_backend.one_
↪qubit_gates,
                                                     two_qubit_gates=qi_backend.two_
↪qubit_gates)
compiler_engines.extend([ResourceCounter()])
engine = MainEngine(backend=qi_backend, engine_list=compiler_engines)

qubits = engine.allocate_qureg(2)
q1 = qubits[0]
q2 = qubits[1]

H | q1
CNOT | (q1, q2)
All(Measure) | qubits

engine.flush()

print('\nMeasured: {0}'.format([int(q) for q in qubits]))
print('Probabilities {0}'.format(qi_backend.get_probabilities(qubits)))
```

ProjectQ example 2

An example that demonstrates how to use the SDK to create a more complex circuit to run Grover's algorithm and simulate the circuit on Quantum Inspire.

```
"""
This example is copied from https://github.com/ProjectQ-Framework/ProjectQ
and is covered under the Apache 2.0 license.
"""
import os
import math

from projectq import MainEngine
from projectq.backends import ResourceCounter, Simulator
from projectq.meta import Compute, Control, Loop, Uncompute
from projectq.ops import CNOT, CZ, All, H, Measure, X, Z
from projectq.setups import restrictedgateset

from quantuminspire.api import QuantumInspireAPI
from quantuminspire.credentials import get_authentication
from quantuminspire.projectq.backend_qx import QIBackend

QI_URL = os.getenv('API_URL', 'https://api.quantum-inspire.com/')
```

(continues on next page)

(continued from previous page)

```
def run_grover(eng, n, oracle):
    """
    Runs Grover's algorithm on n qubit using the provided quantum oracle.

    Args:
        eng (MainEngine): Main compiler engine to run Grover on.
        n (int): Number of bits in the solution.
        oracle (function): Function accepting the engine, an n-qubit register,
            and an output qubit which is flipped by the oracle for the correct
            bit string.

    Returns:
        solution (list<int>): Solution bit-string.
    """
    x = eng.allocate_qureg(n)

    # start in uniform superposition
    All(H) | x

    # number of iterations we have to run:
    num_it = int(math.pi / 4. * math.sqrt(1 << n))

    # prepare the oracle output qubit (the one that is flipped to indicate the
    # solution. start in state 1/sqrt(2) * (|0> - |1>) s.t. a bit-flip turns
    # into a (-1)-phase.
    oracle_out = eng.allocate_qubit()
    X | oracle_out
    H | oracle_out

    # run num_it iterations
    with Loop(eng, num_it):
        # oracle adds a (-1)-phase to the solution
        oracle(eng, x, oracle_out)

        # reflection across uniform superposition
        with Compute(eng):
            All(H) | x
            All(X) | x

        with Control(eng, x[0:-1]):
            Z | x[-1]

        Uncompute(eng)

    All(Measure) | x
    Measure | oracle_out

    eng.flush()
    # return result
    return [int(qubit) for qubit in x]
```

(continues on next page)

(continued from previous page)

```
def alternating_bits_oracle(eng, qubits, output):
    """
    Marks the solution string 1,0,1,0,...,0,1 by flipping the output qubit,
    conditioned on qubits being equal to the alternating bit-string.

    Args:
        eng (MainEngine): Main compiler engine the algorithm is being run on.
        qubits (Qureg): n-qubit quantum register Grover search is run on.
        output (Qubit): Output qubit to flip in order to mark the solution.
    """
    with Compute(eng):
        All(X) | qubits[1::2]
    with Control(eng, qubits):
        X | output
    Uncompute(eng)


# Remote Quantum-Inspire backend
authentication = get_authentication()
qi = QuantumInspireAPI(QI_URL, authentication)
qi_backend = QIBackend(quantum_inspire_api=qi)

compiler_engines = restrictedgateset.get_engine_list(one_qubit_gates=qi_backend.one_
↪qubit_gates,
                                                     two_qubit_gates=qi_backend.two_
↪qubit_gates,
                                                     other_gates=qi_backend.three_qubit_
↪gates)
compiler_engines.extend([ResourceCounter()])
qi_engine = MainEngine(backend=qi_backend, engine_list=compiler_engines)

# Run remote Grover search to find a n-bit solution
result_qi = run_grover(qi_engine, 3, alternating_bits_oracle)
print("\nResult from the remote Quantum-Inspire backend: {}".format(result_qi))

# Local ProjectQ simulator backend
compiler_engines = restrictedgateset.get_engine_list(one_qubit_gates="any", two_qubit_
↪gates=(CNOT, CZ))
compiler_engines.append(ResourceCounter())
local_engine = MainEngine(Simulator(), compiler_engines)

# Run local Grover search to find a n-bit solution
result_local = run_grover(local_engine, 3, alternating_bits_oracle)
print("Result from the local ProjectQ simulator backend: {}\n".format(result_local))
```

1.3.3 Qiskit examples

Qiskit example 1

A simple example that demonstrates how to use the SDK to create a circuit to create a Bell state, and simulate the circuit on Quantum Inspire.

```
"""Example usage of the Quantum Inspire backend with the Qiskit SDK.

A simple example that demonstrates how to use the SDK to create
a circuit to create a Bell state, and simulate the circuit on
Quantum Inspire.

For documentation on how to use Qiskit we refer to
[https://qiskit.org/](https://qiskit.org/).

Specific to Quantum Inspire is the creation of the QI instance, which is used to set the␣
↪authentication
of the user and provides a Quantum Inspire backend that is used to execute the circuit.

Copyright 2018-19 QuTech Delft. Licensed under the Apache License, Version 2.0.
"""
import os

from qiskit import execute
from qiskit.circuit import QuantumRegister, ClassicalRegister, QuantumCircuit

from quantuminspire.credentials import get_authentication
from quantuminspire.qiskit import QI

QI_URL = os.getenv('API_URL', 'https://api.quantum-inspire.com/')


project_name = 'Qiskit-entangle'
authentication = get_authentication()
QI.set_authentication(authentication, QI_URL, project_name=project_name)
qi_backend = QI.get_backend('QX single-node simulator')

q = QuantumRegister(2)
b = ClassicalRegister(2)
circuit = QuantumCircuit(q, b)

circuit.h(q[0])
circuit.cx(q[0], q[1])
circuit.measure(q, b)

qi_job = execute(circuit, backend=qi_backend, shots=256)
qi_result = qi_job.result()
histogram = qi_result.get_counts(circuit)
print('\nState\tCounts')
[print('{0}\t\t{1}'.format(state, counts)) for state, counts in histogram.items()]
# Print the full state probabilities histogram
probabilities_histogram = qi_result.get_probabilities(circuit)
```

(continues on next page)

(continued from previous page)

```
print('\nState\tProbabilities')
[print('{0}\t\t{1}'.format(state, val)) for state, val in probabilities_histogram.
↪items()]
```

Qiskit example 2

A simple example that demonstrates how to use the SDK to create a circuit to demonstrate conditional gate execution.

```
"""Example usage of the Quantum Inspire backend with the Qiskit SDK.

A simple example that demonstrates how to use the SDK to create
a circuit to demonstrate conditional gate execution.

For documentation on how to use Qiskit we refer to
[https://qiskit.org/](https://qiskit.org/).

Specific to Quantum Inspire is the creation of the QI instance, which is used to set the␣
↪authentication
of the user and provides a Quantum Inspire backend that is used to execute the circuit.

Copyright 2018-19 QuTech Delft. Licensed under the Apache License, Version 2.0.
"""
import os

from qiskit import BasicAer, execute
from qiskit.circuit import QuantumRegister, ClassicalRegister, QuantumCircuit

from quantuminspire.credentials import get_authentication
from quantuminspire.qiskit import QI

QI_URL = os.getenv('API_URL', 'https://api.quantum-inspire.com/')


authentication = get_authentication()
QI.set_authentication(authentication, QI_URL)
qi_backend = QI.get_backend('QX single-node simulator')

q = QuantumRegister(3, "q")
c0 = ClassicalRegister(1, "c0")
c1 = ClassicalRegister(1, "c1")
c2 = ClassicalRegister(1, "c2")
qc = QuantumCircuit(q, c0, c1, c2, name="conditional")

qc.h(q[0])
qc.h(q[1]).c_if(c0, 0)  # h-gate on q[1] is executed
qc.h(q[2]).c_if(c1, 1)  # h-gate on q[2] is not executed

qc.measure(q[0], c0)
qc.measure(q[1], c1)
qc.measure(q[2], c2)
```

(continues on next page)

(continued from previous page)

```
qi_job = execute(qc, backend=qi_backend, shots=1024)
qi_result = qi_job.result()
histogram = qi_result.get_counts(qc)
print("\nResult from the remote Quantum Inspire backend:\n")
print('State\tCounts')
[print('{0}\t{1}'.format(state, counts)) for state, counts in histogram.items()]

print("\nResult from the local Qiskit simulator backend:\n")
backend = BasicAer.get_backend("qasm_simulator")
job = execute(qc, backend=backend, shots=1024)
result = job.result()
print(result.get_counts(qc))
```

Back to the *main page*.

1.4 Contributing to Quantum Inspire Examples

We welcome contributions to our Quantum Inspire Examples repository.

By contributing to the repository you state you own the copyright to those contributions and agree to include your contributions as part of this project under the Apache License version 2.0.

For additions, bug fixes or improvements always make a branch first.

After uploading a branch one can make a pull request to the *dev* branch which will be reviewed by our main developers. If the contribution is up to standards we will include it in the QIE repository. We advise to make a sub-directory for each example. A sub-directory is mandatory when the example consist of more than a single file.

1.4.1 Uploading notebooks

Additional notebooks can be added to the /docs/notebooks directory.

1.4.2 Uploading other examples

Additions not in the form of a Jupyter notebook can be added to the /docs/examples directory.

1.4.3 Code style

We follow the PEP 8 style guide. Many editors support autopep8 that can help with coding style. To allow longer lines, make a *.pep8* config file like:

```
[pep8]
max-line-length = 120
```

Use Google Style Python Docstrings for documentation.

1.4.4 Bugs reports and feature requests

If you don't know how to solve a bug yourself or want to request a feature, you can raise an issue via github's issues. Please first search for existing and closed issues, if your problem or idea is not yet addressed, please open a new issue.

CHAPTER

TWO

LICENSE

Apache License

Version 2.0, January 2004

http://www.apache.org/licenses/

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

   "License" shall mean the terms and conditions for use, reproduction, and distribution as defined by Sections 1 through 9 of this document.

   "Licensor" shall mean the copyright owner or entity authorized by the copyright owner that is granting the License.

   "Legal Entity" shall mean the union of the acting entity and all other entities that control, are controlled by, or are under common control with that entity. For the purposes of this definition, "control" means (i) the power, direct or indirect, to cause the direction or management of such entity, whether by contract or otherwise, or (ii) ownership of fifty percent (50%) or more of the outstanding shares, or (iii) beneficial ownership of such entity.

   "You" (or "Your") shall mean an individual or Legal Entity exercising permissions granted by this License.

   "Source" form shall mean the preferred form for making modifications, including but not limited to software source code, documentation source, and configuration files.

   "Object" form shall mean any form resulting from mechanical transformation or translation of a Source form, including but not limited to compiled object code, generated documentation, and conversions to other media types.

   "Work" shall mean the work of authorship, whether in Source or Object form, made available under the License, as indicated by a copyright notice that is included in or attached to the work (an example is provided in the Appendix below).

   "Derivative Works" shall mean any work, whether in Source or Object form, that is based on (or derived from) the Work and for which the editorial revisions, annotations, elaborations, or other modifications represent, as a whole, an original work of authorship. For the purposes of this License, Derivative Works shall not include works that remain separable from, or merely link (or bind by name) to the interfaces of, the Work and Derivative Works thereof.

   "Contribution" shall mean any work of authorship, including the original version of the Work and any modifications or additions to that Work or Derivative Works thereof, that is intentionally submitted to Licensor for inclusion in the Work by the copyright owner or by an individual or Legal Entity authorized to submit on behalf of the copyright owner. For the purposes of this definition, "submitted" means any form of electronic, verbal, or written communication sent to the Licensor or its

representatives, including but not limited to communication on electronic mailing lists, source code control systems, and issue tracking systems that are managed by, or on behalf of, the Licensor for the purpose of discussing and improving the Work, but excluding communication that is conspicuously marked or otherwise designated in writing by the copyright owner as "Not a Contribution."

"Contributor" shall mean Licensor and any individual or Legal Entity on behalf of whom a Contribution has been received by Licensor and subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:

   (a) You must give any other recipients of the Work or Derivative Works a copy of this License; and

   (b) You must cause any modified files to carry prominent notices stating that You changed the files; and

   (c) You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of the Derivative Works; and

   (d) If the Work includes a "NOTICE" text file as part of its distribution, then any Derivative Works that You distribute must include a readable copy of the attribution notices contained within such NOTICE file, excluding those notices that do not pertain to any part of the Derivative Works, in at least one of the following places: within a NOTICE text file distributed as part of the Derivative Works; within the Source form or documentation, if provided along with the Derivative Works; or, within a display generated by the Derivative Works, if and wherever such third-party notices normally appear. The contents of the NOTICE file are for informational purposes only and do not modify the License. You may add Your own attribution notices within Derivative Works that You distribute, alongside or as an addendum to the NOTICE text from the Work, provided that such additional attribution notices cannot be construed as modifying the License.

   You may add Your own copyright statement to Your modifications and may provide additional or different license terms and conditions for use, reproduction, or distribution of Your modifications, or for any such Derivative Works as a whole, provided Your use, reproduction, and distribution of the Work otherwise complies with the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise, any Contribution intentionally submitted for inclusion in the Work by You to the Licensor shall be under the terms and conditions of this License, without any additional terms or conditions. Notwithstanding the above, nothing herein shall supersede or modify the terms of any separate license agreement you may have executed with Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade names, trademarks, service marks, or product names of the Licensor, except as required for reasonable and customary use in describing the origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or agreed to in writing, Licensor provides the Work (and each Contributor provides its Contributions) on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied, including, without limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A PARTICULAR PURPOSE. You are solely responsible for determining the appropriateness of using or redistributing the Work and assume any risks associated with Your exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory, whether in tort (including negligence), contract, or otherwise, unless required by applicable law (such as deliberate and grossly negligent acts) or agreed to in writing, shall any Contributor be liable to You for damages, including any direct, indirect, special, incidental, or consequential damages of any character arising as a result of this License or out of the use or inability to use the Work (including but not limited to damages for loss of goodwill, work stoppage, computer failure or malfunction, or any and all other commercial damages or losses), even if such Contributor has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability. While redistributing the Work or Derivative Works thereof, You may choose to offer, and charge a fee for, acceptance of support, warranty, indemnity, or other liability obligations and/or rights consistent with this License. However, in accepting such obligations, You may act only on Your own behalf and on Your sole responsibility, not on behalf of any other Contributor, and only if You agree to indemnify, defend, and hold each Contributor harmless for any liability incurred by, or claims asserted against, such Contributor by reason of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS