Quantum Computing Functions (QCF) for Matlab

Charles Fox
Robotics Research Group
Oxford University

Quantum computing uses unitary operators acting on discrete state vectors. Matlab is a well known (classical) matrix computing environment, which makes it well suited for simulating quantum algorithms. The QCF library extends Matlab by adding functions to represent and visualize common quantum operations. This paper presents a brief overview of QCF, then shows how it can be used to simulate the well-known algorithms of Deutsch, Deutch-Jozsa, and Grover.

Basic notation

There are three basic quantum state notations that are frequently used in the literature: integer kets, binary kets, and vectors. For example, the following are all representations of the same state (in a 3-qubit space):

|3> |011> $[0\ 0\ 0\ 1\ 0\ 0\ 0\ 0\ ]^T$

The simulator uses vectors as its primary internal representation, but provides functions to convert between notations:

phi = bin2vec(bin)
Converts a single binary string state representation to a state vector.

phi=dec2vec(dec,n)
Convert single decimal state representation to state vector.
n is number of qubits in the vector space.

str = pretty(psi, [bin])
Gives a pretty-printed ket string for a (possibly superposed) state vector *psi*.
By default the integer ket representation is used. The optional *bin* flag gives the binary version.

For example:

```
>> phi_1=bin2vec('011')

     0
     0
     0
     1
     0
     0
     0
     0

>> phi_2 = dec2vec(5, 3)

     0
     0
     0
     0
     0
     1
     0
     0
```

Note that our vector representation is capable of handling superposed states:

```
>> psi = 1/sqrt(2)*phi_1 + 1/sqrt(2)*phi_2

         0
         0
         0
    0.7071
         0
    0.7071
         0
         0
```

…and so are the pretty-print functions:

```
>> pretty(psi)

0.7071|011> + 0.7071|101>

>> pretty(psi, 1)

0.7071|3> + 0.7071|5>
```

Fourier transforms

The Quantum Fourier Transform (QFT) uses *normalized* basis functions (unlike the classical Discrete Fourier Transform) to represent a discrete state vector:

$$|x> = \frac{1}{\sqrt{N}} \Sigma_{j=1:N-1} e^{-2\pi ixj/N} |j>$$

As the basis is orthonormal, the QFT projections can be computed by the unitary transform:

$$\text{QFT} = \frac{1}{N} \begin{bmatrix} \omega^0 & \omega^1 & \omega^2 & \omega^3 & . \\ \omega^1 & \omega^2 & \omega^3 & \omega^4 & . \\ \omega^2 & \omega^3 & \omega^4 & \omega^5 & . \\ \omega^3 & \omega^4 & \omega^5 & \omega^6 & \\ \dots & & & & \dots \end{bmatrix}$$

... where ω is the N^{th} root of unity, $e^{2\pi i/N}$.

The QCF library provides a *qft* function which creates QFT matrices for any *N*:

QFT = qft(d)
Creates QFT matrix with *d* rows and cols, where $d=2^N$.

Note that the QFT is both unitary and hermetian, so the matrix obtained from the *qft* function can be used to transform in both directions.

In the following example we create a QFT matrix for a 3-qubit system:

```
>> Q = qft(2^3)


  Columns 1 through 4

   0.3536             0.3536             0.3536             0.3536
   0.3536             0.2500 + 0.2500i   0.0000 + 0.3536i  -0.2500 + 0.2500i
   0.3536             0.0000 + 0.3536i  -0.3536 + 0.0000i  -0.0000 - 0.3536i
   0.3536            -0.2500 + 0.2500i  -0.0000 - 0.3536i   0.2500 + 0.2500i
   0.3536            -0.3536 + 0.0000i   0.3536 - 0.0000i  -0.3536 + 0.0000i
   0.3536            -0.2500 - 0.2500i   0.0000 + 0.3536i   0.2500 - 0.2500i
   0.3536            -0.0000 - 0.3536i  -0.3536 + 0.0000i   0.0000 + 0.3536i
   0.3536             0.2500 - 0.2500i  -0.0000 - 0.3536i  -0.2500 - 0.2500i

  Columns 5 through 8

   0.3536             0.3536             0.3536             0.3536
  -0.3536 + 0.0000i  -0.2500 - 0.2500i  -0.0000 - 0.3536i   0.2500 - 0.2500i
   0.3536 - 0.0000i   0.0000 + 0.3536i  -0.3536 + 0.0000i  -0.0000 - 0.3536i
  -0.3536 + 0.0000i   0.2500 - 0.2500i   0.0000 + 0.3536i  -0.2500 - 0.2500i
   0.3536 - 0.0000i  -0.3536 + 0.0000i   0.3536 - 0.0000i  -0.3536 + 0.0000i
  -0.3536 + 0.0000i   0.2500 + 0.2500i  -0.0000 - 0.3536i  -0.2500 + 0.2500i
   0.3536 - 0.0000i  -0.0000 - 0.3536i  -0.3536 + 0.0000i   0.0000 + 0.3536i
  -0.3536 + 0.0000i  -0.2500 + 0.2500i   0.0000 + 0.3536i   0.2500 + 0.2500i
```

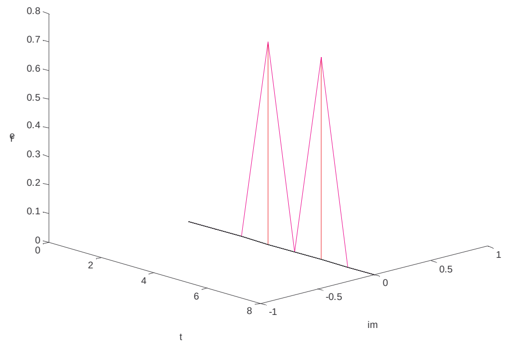
We now apply the QFT to the *psi* state from the earlier example:

```
>> pretty(psi, 1)

0.7071|3> + 0.7071|5>


>> QFT*psi

   0.5000
  -0.3536 + 0.0000i
   0.0000
   0.3536 - 0.0000i
  -0.5000 + 0.0000i
   0.3536 - 0.0000i
  -0.0000
  -0.3536 + 0.0000i

>> pretty(QFT*psi)

0.5|000> + -0.35355+1.3878e-016i|001> + 1.1102e-016|010> + 0.35355-4.7184e-016i|011> + -
0.5+8.8817e-016i|100> + 0.35355-7.7716e-016i|101> + -3.3306e-016|110> + -0.35355+1.0825e-
015i|111>
```

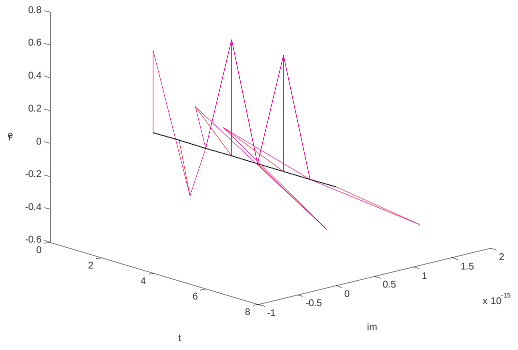
Our QCF library includes the function *iplot* for visualizing complex vectors. In the example below, we use *iplot* to look at the effect of the QFT on *psi*:

> **iplot(psi)**
> Display a 3D complex plane graph of the complex vector *psi*.
> The elements of *psi* are shown as dark spikes and are connected together by a lighter line.

```
>> iplot(psi)
```



```
>> iplot(QFT*psi)
```



Unitary matrix representation of functions

It is often useful to use unitary matrices to represent functions. In this representation, the functions can be performed 'in parallel' on superpositions of inputs. We define the matrix U_f by:

$$U_f\,(\,|x>\otimes|0>^{(n)}\,) = |x>\otimes|f(x \bmod n)>$$

Where there are m bits in the input, x, and n bits in the output. Note that the action of U_f is to replace the zero vector with the value of $f(x)$, whilst leaving the x in tact.

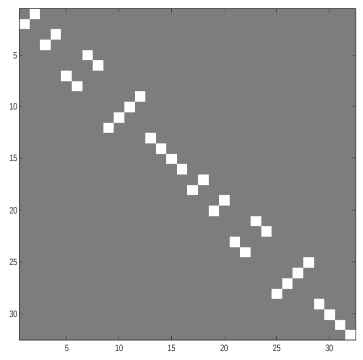
The QCF library function *uf* provides a quick way to construct such unitary matrices. To use it, we define an ordinary function *f*, and pass it as an argument to the *uf* constructor, along with specifications of *n* and *m*. This is necessary because Matlab functions operate on integers, not binary strings – so *uf* must be told how many bits to make space for.

U_f = uf(f,m,n)
Make unitary U_f from function *f*.
f must be of the form $f(x,n)$ where x is a bitstring, and m and n is numbers of input and output bits.

Here we define a simple function, *y*, which returns the modulus of *x*+1. We then display a visualization of the unitary U_y which represents it:

```
function y = f(x,n)
y=mod(x+1, 2^n);
```

```
>> U_f = uf('f', 3, 2);
>> qimage(U_f)
```



We now illustrate how the U_f matrix can be used. We form the state |*psi*> ⊗ |0>$^{(2)}$ using the *kron* function for the tensor product. Applying U_f to the result simultaneously computes the superposed results of *f* for the two superposed states comprising |*psi*>.

```
>> pretty(psi)
0.7071|011> + 0.7071|101>

>> pretty(bin2vec('00'))
1|00>

>> pretty(kron(psi, bin2vec('00')))
0.7071|01100> + 0.7071|10100>

>> pretty(U_f * kron(psi, bin2vec('00')))
0.7071|01100> + 0.7071|10110>
```

Deutsch's algorithm

Deutsch's was one of the first quantum algorithms to demonstrate an effect unobtainable by classical computation. Suppose we have four functions: c_0, c_1, b_0 and b_1 as given below:

Input:	0	1
c_0	0	0
c_1	1	1
b_0	0	1
b_1	1	0

The *c* functions are said to be 'constant' and the *b* functions are 'balanced'. Given a black-box which computes U_f, where f is a function chosen from the above set, the task is to determine whether the function represented by U_f is constant or balanced. A classical machine would require two evaluations of the black-box. But Deutsch's algorithm provides a way to do it in just one evaluation, on a quantum computer. The algorithm makes use the Hadamard transform, implemented in QCF by:

H = hadamard(n)
Returns the *n*-qubit Hadamard matrix.

We also use the *measure* function to collapse the final superposition into a single state:

phi = measure(psi)
Measure psi with respect to the standard basis.
(To perform other measurements, transform *psi* and *phi* to and from the required basis.)

The code below implements Deutsch for f=b_1. The panel on the right shows the variables during the execution. The results show the values of *psi* at the end of the algorithm, for the four black-box functions. Looking at the first qubit of the final states tells us whether the function is constant (0) or balanced (1).

```
function deutch
psi = bin2vec('01')
U_f = uf('f_b1', 1, 1)
H = hadamard(2)
psi = H*U_f*H*psi
psi = measure(psi)

function y = f_c0(x,n)
y=0;

function y = f_c1(x,n)
y=1;

function y = f_b0(x,n)
y=x;

function y = f_b1(x,n)
y=~x;
```

Results:

function		final *psi*
b_0	à	|11>
b_1	à	|11>
c_0	à	|01>
c_1	à	|01>

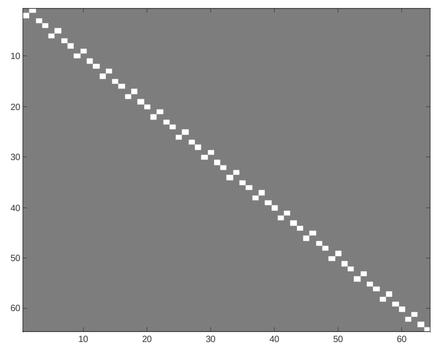
```
psi (initial value) =
   0
   1
   0
   0

U_f =
   0   1   0   0
   1   0   0   0
   0   0   1   0
   0   0   0   1

H =
   0.5000    0.5000    0.5000    0.5000
   0.5000   -0.5000    0.5000   -0.5000
   0.5000    0.5000   -0.5000   -0.5000
   0.5000   -0.5000   -0.5000    0.5000

psi (just before measurement) =
        0
        0
        0
  -1.0000

psi (after measurement) =
   0
   0
   0
   1
```

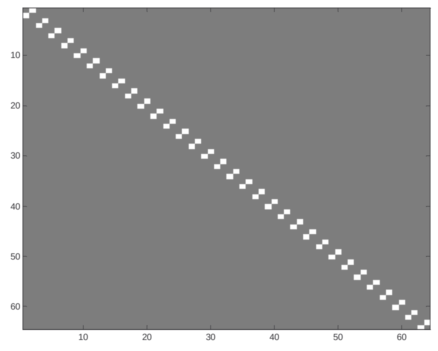
The Deutsch -Jozsa Promise Problem

This is a generalization of Deutsch's algorithm to cases where the inputs to the constant/balanced functions have multiple bits. The functions output either 0 or 1, and are 'promised' to be either constant or balanced. Constant now means that the output is the same for any possible input. Balanced means that there are equal numbers of inputs which output 0 as output 1. As with the original Deutsch algorithm, the Deutsch-Jozsa problem can be solved with just one black box evaluation of the U_f.

Matlab/QCF code for Deutsch-Jozsa and some example test functions are shown below. Note the *deutche_joza* function has been defined so that a test function is passed in as an argument. The final, measured, *psi* state is interpreted by looking at the first *N*-1 qubits. If they are all zero then the function is constant. Otherwise, it is balanced. The images on the right are visualizations of two of the U_f matrices.

```
function deutsch_jozsa(f)
psi = bin2vec('000001');
U_f = uf(f, 5, 1);
H_6 = hadamard(6);
H_5 = hadamard(5);
I   = identity(1);
psi = (kron(H_5, I))*U_f*H_6*psi;
psi = measure(psi);
pretty(psi)



%the test functions:

function y = c_0(x,n)
y=1;
function y = c_1(x,n)
y=0;
function y = b_0(x,n)
y=x;
function y = b_1(x,n)
y=~x;



>> deutch_joza('c_0')
 1|000000>

>> deutch_joza('c_1')
1|000001>

>> deutch_joza('b_1')
1|000010>

>> deutch_joza('b_0')
1|000011>
```


`U_f` for `b_0`


`U_f` for `c_0`

By looking a the first four qubits, we see that
So c_0 and c_1 are constant, b_0 and b_1 are balanced.

Grover's Algorithm

Grover demonstrated that quantum computers can perform database search faster than their classical counterparts. In this simple example of Grover's algorithm, we use a function *haystack* to represent the database. We are searching for a 'needle in the haystack', i.e. there is one element of the database that we require. The *haystack* function returns 1 if queried with the correct 'needle' element, and 0 for all the other elements. In this example, *haystack* is defined for a 5-qubit input, and returns 1 for element 8 ('00100').

Grover makes use of the fact that for any function *f* with a binary string input and a Boolean output, we can define a corresponding unitary matrix V_f, whose action on input strings |*psi*> is to invert them if and only if f(*psi*)=1. (Otherwise, it does nothing). The QCF library provides a function to create such matrices:

V_f = vf(f, n)
Build unitary *f*-conditional inverter for *n* bit input such that:
$V_f|psi\rangle = (-1)^{f(psi)}|psi\rangle$

Additionally, an 'inversion about the average' matrix, D, is used. The action of D on each element a_i of a state vector *psi* is to replace it with $\text{mean}_j(a_j)-a_i$. Again, QCF has a function to build D for a specified state vector size.

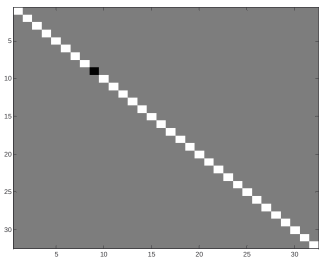
D = ia(n)
Create 'inversion about average' matrix for *n*-qubit state vectors.

Grover's algorithms works by iteratively applying $V_{haystack}$ and the inversion about the average operator to the current state. Each iteration amplifies the probability of a measurement collapsing the state to the correct 'needle' value.
Grover showed that performing a measurement after $\sqrt{(2^n)}*(\pi/4)$ iterations is highly likely to give the correct result.

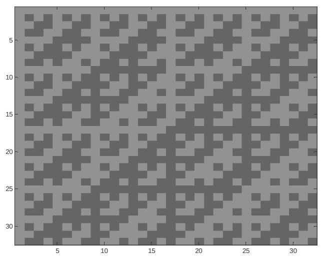
The code for Grover's algorithm is given below.
On the right are visualizations of the matrices used.
The 3D graph on the next page shows the amplitudes of the different 'needle' candidates. Note that the amplitude for 8 (the correct needle value) reaches its peak at about $\sqrt{(2^n)}*(\pi/4)$ iterations, as predicted.

```
function grover

n=5;

V_haystack  = vf('haystack', n);
H             = hadamard(n);
D             = ia(n);

phi = bin2vec('00000');

phi = H*phi;

maxiter = (pi/4)*sqrt(2^n);

for i=1:10

    phi=V_haystack*phi;
    phi=D*phi;

end

phi=measure(phi);
```

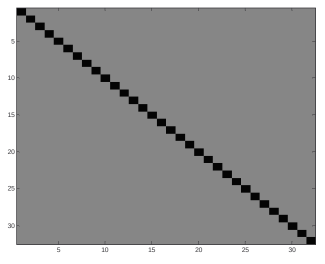
(black=-1; gray=0; white=1+)



$V_{haystack}$

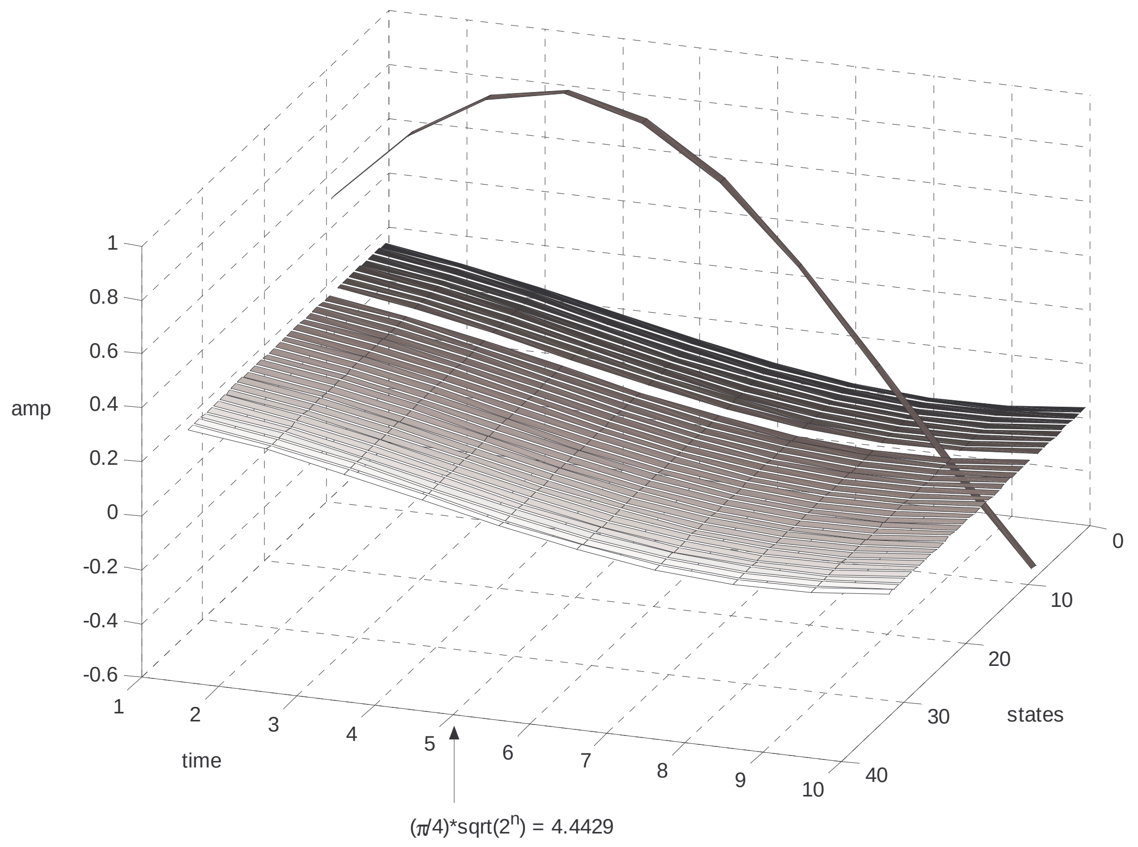


H



D

Amplitudes for needle values during Grover's algorithm iterations:



QCF API Summary

phi = bin2vec(bin)
Converts a single binary string state representation to a state vector.

phi=dec2vec(dec,n)
Convert single decimal state representation to state vector.
n is number of qubits in the vector space.

str = pretty(psi, [bin])
Gives a pretty-printed ket string for a (possibly superposed) state vector *psi*.
By default the integer ket representation is used. The optional *bin* flag gives the binary version.

QFT = qft(d)
Creates QFT matrix with *d* rows and cols, where $d=2^N$.

iplot(psi)
Display a 3D complex plane graph of the complex vector *psi*.
The elements of *psi* are shown as dark spikes and are connected together by a lighter line.

H = hadamard(n)
Returns the *n*-qubit Hadamard matrix.

phi = measure(psi)
Measure psi with respect to the standard basis.
(To perform other measurements, transform *psi* and *phi* to and from the required basis.)

V_f = vf(f, n)
Build unitary *f*-conditional inverter for *n* bit input such that:
$V_f|psi> = (-1)^{f(psi)}|psi>$

D = ia(n)
Create ‘inversion about average’ matrix for *n*-qubit state vectors.