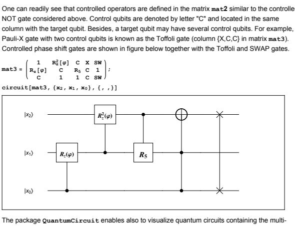
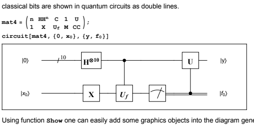


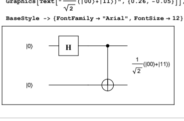
The package **QuantumCircuit** enables also to visualize quantum circuits containing the multi-qubit wires and classical bits. For example, quantum circuit shown in figure below contains *n*-qubit wire (symbol "n" in matrix **mat4**) and each qubit in this wire is acted on by the Hadamard gate, the corresponding operator is shown at the diagram as a tensor product of *n* Hadamard gates and is denoted in **mat4** by symbol HH^n. Then this wire is used to control operator U_f acting on qubit $|x_0\rangle$ (symbol U_f in matrix **mat4**). After measurement of qubit $|x_0\rangle$ a classical bit $|f_0\rangle$ is obtained that is used afterwards to control some unitary operator *U* acting on the state of *n*-qubit wire. Note that classical bits are shown in quantum circuits as double lines.

```
In[64]:= mat4 = (n  HH^n  C  1  U
                 1  X     Uf M  CC);
circuit[mat4, {0, x0}, {y, f0}]
```

Out[65]=


Using function **Show** one can easily add some graphics objects into the diagram generated by function **circuit**. The corresponding command is shown below.

```
In[66]:= mat5 = (H C
                 1 X);
Show[circuit[mat5, {0, 0}, {, }],
 Graphics[Text["1/√2 (|00⟩+|11⟩)", {0.26, -0.05}]],
 BaseStyle -> {FontFamily → "Arial", FontSize → 12} ]
```

Out[67]=


Unitary matrix corresponding to quantum circuit

Using the function **basisState** defined above one can easily construct an abitrary vector in the 2^n-dimentional state space, corresponding to an initial state of the *n*-qubit memory register. Then this state should be transformed to a final state by means of a unitary transformation determined by a quantum circuit. The corresponding $2^n \times 2^n$ unitary matrix is computed by the function **matrixU[mat]**, its argument **mat** is a symbolic matrix encoding information about the quantum circuit. Note that this function calculates unitary matrices corresponding to each column of symbolic matrix **mat** and the final unitary matrix ***U*** is obtained as a product

$U = U_m . U_{m-1} \ldots U_1$

where U_j ($j = 1, 2, \ldots, m$) is the $2^n \times 2^n$ matrix determined by quantum gates being in the *j*th column of the matrix **mat**, and *m* is a number of columns in **mat**. Matrix representation of the Pauli-X and Toffoli gates are given by

```
In[68]:= matrixU[{{X}}] // MatrixForm
```

Out[68]//MatrixForm=
$$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

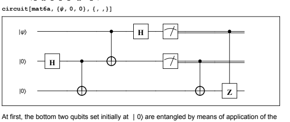
```
In[69]:= matrixU[{{C}, {C}, {X}}] // MatrixForm
```

Out[69]//MatrixForm=
$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

Circuit implementing quantum teleportation

To demonstrate usage of the function **matrixU**, let us consider quantum circuit that is used in quantum teleportation. The problem is to transmit an arbitrary state $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$ of the top qubit to the bottom qubit. The corresponding quantum circuit is shown below.

```
In[70]:= mat6a = (1 1 C H M c1 CC
                  H C X 1 M CC c1
                  1 X 1 1 1 X  Z);
circuit[mat6a, {ψ, 0, 0}, {, ,}]
```

Out[71]=


At first, the bottom two qubits set initially at $|0\rangle$ are entangled by means of application of the Hadamard and CNOT gates. Then two qubits are measured and the classical bits obtained are transmitted to the receiver. Then two qubits are measured and CNOT and controlled Pauli-Z gate are applied to the third qubit.

Initial state $|\psi 00\rangle$ of the three-qubit memory register is defined as a superposition of the two basis states $|000\rangle$ and $|100\rangle$, the corresponding vector is given by

```
In[6]:= vecInput = α basisState[FromDigits[{0, 0, 0}, 2], 3] +
    β basisState[FromDigits[{1, 0, 0}, 2], 3];
```

To find a final state we multiply this vector by the unitary matrix corresponding to the circuit that is computed by the function **matrixU**.

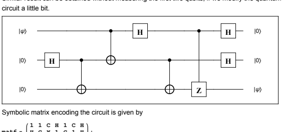
```
In[7]:= mat6b = (1 1 C H
                 H C X 1
                 1 X 1 1);
vecOutputB = matrixU[mat6b].vecInput // Normal
```

Out[8]= $\left\{\frac{\alpha}{2}, \frac{\beta}{2}, \frac{\beta}{2}, \frac{\alpha}{2}, \frac{\alpha}{2}, -\frac{\beta}{2}, -\frac{\beta}{2}, \frac{\alpha}{2}\right\}$

To save memory we use sparse arrays for representing unitary matrices determined by quantun circuits. So the function **Normal** is used to represent the final vector in the normal form.
This vector can be rewritten in the form

$$|00\rangle \frac{\alpha|0\rangle + \beta|1\rangle}{2} + |01\rangle \frac{\alpha|1\rangle + \beta|0\rangle}{2} + |10\rangle \frac{\alpha|0\rangle - \beta|1\rangle}{2} + |11\rangle \frac{\alpha|1\rangle - \beta|0\rangle}{2}$$

Similar result can be obtained without measuring the first two qubits, if we modify the quantum circuit a little bit.



Symbolic matrix encoding the circuit is given by

```
In[9]:= mat6 = (1 1 C H 1 C H
                H C X 1 C 1 H
                1 X 1 1 X Z 1);
In[10]:= vecOutput = matrixU[mat6].vecInput // Normal
```

Out[10]= {α, β, 0, 0, 0, 0, 0, 0}

The following commands enable to represent the final vector as a superposition of basis states.

```
In[11]:= vecOutBinary[α_] := Map[IntegerDigits[#[[1]] - 1, 2, 3] &, Position[vecOutput, α]];
coeffOutput[α_] := Cases[vecOutput, _. α] /. x_. α → x;
braInput[α_] := Apply[Plus, Table[ coeffOutput[α][[k]] (vecOutBinary[α][[k]] //
        StringJoin["|", Map[ToString[#] &, vecOutBinary[α][[k]] ], "⟩"] &),
    {k, Length[vecOutBinary[α]]}] ]
Apply[Plus, {α braInput[α], β braInput[β]}]
```

Out[14]= |000⟩ α + |001⟩ β

One can readily see that the final state obtained is exactly the state

$|000\rangle \alpha + |001\rangle \beta = |00\rangle (\alpha|0\rangle + \beta|1\rangle) = |00\rangle|\psi\rangle = |00\psi\rangle$.

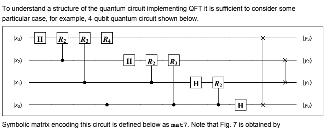
Thus, using symbolic matrix for representation of quantum circuit we can easily visualize the circuit, calculate the corresponding unitary matrix and find a final state of the memory register for any given initial state. Note that constructing a quantum circuit in this approach is very easy and reduces to defining a symbolic matrix encoding information about the circuit.

Implementation of quantum Fourier transform

It should be emphasized that symbolic matrix encoding information about quantum circuit may be generated, as well, if algorithm of the circuit construction is given. To demonstrate such an example, let us consider the quantum Fourier transform (QFT) that plays a principal role in the development of efficient quantum algorithms (see [C,G]). It is a unitary transformation whose action on the computational basis state is given by

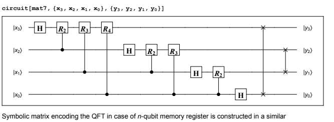
$$U_{FT}|x\rangle_n = \frac{1}{2^{n/2}} \sum_{y=0}^{2^n-1} \exp\left(2\pi i \frac{x y}{2^n}\right) |y\rangle_n$$

To understand a structure of the quantum circuit implementing QFT it is sufficient to consider some particular case, for example, 4-qubit quantum circuit shown below.



Symbolic matrix encoding this circuit is defined below as **mat7**. Note that Fig. 7 is obtained by means of applying the function **circuit**.

```
In[15]:= mat7 = (H R2 R3 R4 1 1  1  1 1  1 SW 1
                 1 C  1  1  H R2 R3 1 1  1 1  SW
                 1 1  C  1  1 C  1  H R2 1 1  SW
                 1 1  1  C  1 1  C  1 C  H SW 1);
circuit[mat7, {x3, x2, x1, x0}, {y3, y2, y1, y0}]
```

Out[15]=


Symbolic matrix encoding the QFT in case of *n*-qubit memory register is constructed in a similar way. To generate such a matrix we define a function **modelFourier**.

```
In[16]:= modelFourier[n_] := Module[{model, mm, n1},
   model = Array[mm, {n, n (n + 1) / 2 + Floor[n / 2]}] /. mm[i_, j_] → 1;
        n1 = 0;
   Do[     Do[ If[k == 1, model[[j, n1 + 1]] = H,
            (model[[j, n1 + k]] = Rk; model[[j + k - 1, n1 + k]] = C)
           ],  {k, n - j + 1}];
      n1 = n1 + n - j + 1,    {j, n}];
   Do[ model[[j, n1 + j]] = SW;
    model[[n - j + 1, n1 + j]] = SW , {j, Floor[n / 2]}];
   model ]
```

One can readily check that symbolic matrix given by function **modelFourier** in case of four qubits coincides with matrix **mat7** defined above.

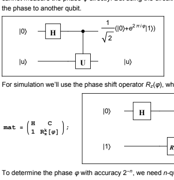
```
In[17]:= modelFourier[4] // MatrixForm
```

Using the same symbolic matrix one can compute the unitary matrix corresponding to the QFT, in case of three qubits it is given by

```
In[18]:= matrixU[modelFourier[3] ] // MatrixForm
```

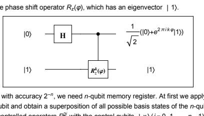
Quantum algorithm for phase estimation

Consider a unitary operator *U* with the eigenvector $|u\rangle$ and the eigenvalue $e^{2\pi i \varphi}$, where φ is an unknown real number, $0 \le \varphi < 1$, which is called the phase. We assume that the quantum circuit implementing the operator *U* is given and the quantum memory register can be set to the state $|u\rangle$. The problem is to construct a quantum circuit for determining the unknown phase φ. Note that we cannot measure the phase φ directly. But using the circuit below we can transfer information about the phase to another qubit.

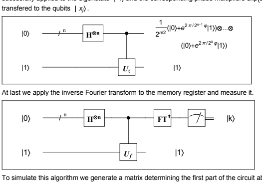


For simulation we'll use the phase shift operator $R_z(\varphi)$, which has an eigenvector $|1\rangle$.

```
mat = (H C
       1 Rz^k[φ]);
```



To determine the phase φ with accuracy 2^{-n}, we need *n*-qubit memory register. At first we apply the Hadamard gate to each qubit and obtain a superposition of all possible basis states of the *n*-qubit memory register. Then *n* controlled operators $R_z^{2^j}$ with the control qubits $|x_j\rangle$ ($j = 0, 1, \ldots, n-1$) are succesfully applied to the eigenstate $|1\rangle$ and the corresponding phase multipliers $\exp(2\pi i 2^j \varphi)$ are transfered to the qubits $|x_j\rangle$.



At last we apply the inverse Fourier transform to the memory register and measure it.

To simulate this algorithm we generate a matrix determining the first part of the circuit above.

```
In[19]:= mat[n_, φ_] := Block[{mat1},
     mat1 = Table[If[j == 1, H,
              If[j == n + 2 - i, C, 1]],
            {i, n}, {j, n + 1}];
     mat1 = Append[mat1,
       Table[If[j == 1, 1, Rz^(2^(j-2)) [φ]],
                           {j, n + 1}]];
     mat1]
```

In case of 5-qubit memory register the corresponding quantum circuit is shown below.

```
In[20]:= circuit[mat[5, φ], {0, 0, 0, 0, 0, 1}, {, , , , ,}]
```

Computing a unitary matrix correcponding to this circuit we transform initial state of the memory register to the state *vec2*.

```
In[21]:= vec1[n_] := matrixU[mat[n, φ]].basisState[1, n + 1] // Normal;
In[22]:= vec2 = Table[vec1[5][[2 i]], {i, 2^5}]
```

Then we apply the inverse Fourier transform and obtain the vector *vec3*.

```
In[23]:= vec3 = Transpose[Conjugate[matrixU[modelFourier[5] ]]].vec2 // Normal ;
```

One can readily check that if the phase φ may be represented in the form of *n*-bit binary fraction a final state of the memory register turns out to be one of the basis states and an exact value of φ is obtain with unit probability as a result of measurement.

```
In[24]:= vec3 /. φ → 17 / 32 // Simplify
```

Out[24]= {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}

If the phase φ ia not a *n*-bit binary fraction then any basis state of the *n*-qubit memory register can be obtained as a result of measurement but probabilities of getting these states are different. In case of $\varphi = 2/3$, for example, the final vector is

```
In[25]:= vec4 = N[Abs[#]^2] & /@ (vec3 /. φ → 2 / 3)
```

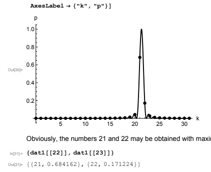
Out[25]= {0.000976563, 0.000882784, 0.000816818, 0.000772241, 0.000745116,
 0.000733207, 0.000735568, 0.000752386, 0.000785007, 0.000836174,
 0.000910541, 0.00101564, 0.00116367, 0.00137492, 0.00168475,
 0.00215875, 0.00292969, 0.00430009, 0.00708864, 0.0142042, 0.0429899,
 0.684162, 0.171224, 0.0276022, 0.0109337, 0.00590282, 0.00374412,
 0.00262653, 0.00197636, 0.00156735, 0.00129572, 0.00110852}

Analysis of the algorithm shows that probability to get a number *y* as a result of measurement is determined by the function

```
In[26]:= p1[n_, y_, φ_] := 1/2^(2 n) Sin[π (2^n φ - y)]^2 / Sin[π (φ - y / 2^n)]^2
```

One can readily see that probabilities of getting different results determined by the vector *vec4* coincide with the corresponding values of the function *p1*.

```
In[27]:= pa = Plot[p1[5, y, 2 / 3], {y, 0, 2^5 - 1},
    PlotStyle → {Thick, Black}, PlotRange → All];
dat1 = Table[{i - 1, vec4[[i]]}, {i, Length[vec4]}];
pb = ListPlot[dat1, PlotRange → All, PlotStyle → {Black, PointSize[0.02]}];
Show[pa, pb, BaseStyle → {FontFamily → "Arial", FontSize → 12},
 AxesLabel → {"k", "p"}]
```

Out[30]=


Obviously, the numbers 21 and 22 may be obtained with maximum probability.

```
In[31]:= {dat1[[22]], dat1[[23]]}
```

Out[31]= {{21, 0.684162}, {22, 0.171224}}

Total probability is greater than $8/\pi^2$.

```
In[33]:= {dat1[[22, 2]] + dat1[[23, 2]], 8. / π^2}
```

Out[33]= {0.855386, 0.810569}

Accuracy of the calculation is given by

```
In[35]:= {2 / 3 - 21. / 32, 2 / 3 - 22. / 32, 1. / 2^6}
```

Out[35]= {0.0104167, -0.0208333, 0.015625}

Obviously, the probability to get any possible result is equal to 1.

```
In[36]:= Apply[Plus, vec4]
```

Out[36]= 1.

To increase an accuracy of calculation let us consider 10-qubit memory register.

```
In[37]:= vec1a = vec1[10];
vec2 = Table[vec1a[[2 i]], {i, 2^10}];
In[38]:= vec3 = Transpose[Conjugate[matrixU[modelFourier[10] ]]].vec2 // Normal ;
```

In case of *n*-bit binary fraction an exact result is obtained with unit probability.

```
In[40]:= vec3 /. φ → 17 / 32 // Simplify
In[41]:= Position[%, 1]
```

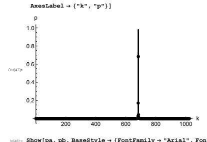
Out[41]= {{545}}

```
In[42]:= 544 / 2^10
```

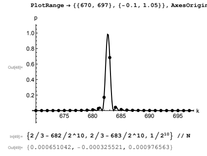
Out[42]= $\frac{17}{32}$

In a general case of $\varphi = 2/3$, for example, we obtain

```
In[43]:= vec4 = N[Abs[#]^2] & /@ (vec3 /. φ → 2 / 3);
pa = Plot[p1[10, y, 2 / 3], {y, 0, 2^10 - 1},
    PlotStyle → {Thick, Black}, PlotRange → All, PlotPoints → 100];
dat1 = Table[{i - 1, vec4[[i]]}, {i, Length[vec4]}];
pb = ListPlot[dat1, PlotRange → All, PlotStyle → {Black, PointSize[0.02]}];
Show[pa, pb, BaseStyle → {FontFamily → "Arial", FontSize → 12},
 AxesLabel → {"k", "p"}]
```

Out[47]=


```
In[48]:= Show[pa, pb, BaseStyle → {FontFamily → "Arial", FontSize → 12},
 AxesLabel → {"k", "p"},
 PlotRange → {{670, 697}, {-0.1, 1.05}}, AxesOrigin → {670, 0}]
```

Out[48]=


```
In[49]:= {2 / 3 - 682 / 2^10, 2 / 3 - 683 / 2^10, 1 / 2^10} // N
```

Out[49]= {0.000651042, -0.000325521, 0.000976563}

```
In[50]:= {dat1[[683, 2]] + dat1[[684, 2]], 8. / π^2}
```

Out[50]= {0.854898, 0.810569}

Approximate QFT

Approximate QFT was first introduced by Barenco, et al [A. Barenco, A. Ekert, K. Suominen, P. Törmä, Approximate Quantum Fourier Transform and Decoherence, Phys. Rev. A, vol. 54, issue 1, pp. 139-146]. It is constructed similar to regular QFT but differs by eliminating higher precision phase shift operators. Let *m* be the degree of the AQFT, it means that the corresponding quantum circuit contains only phase shift operators $R_2, R_3, \ldots, R_m$. One can easilly generate symbolic matrix determining the AQFT of degree *m*, the corresponding function is given by

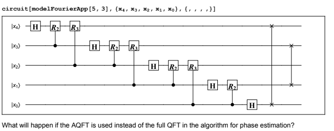
```
In[51]:= modelFourierApp[n_, m_] := Module[{model, mm, n1, m1},
   m1 = 0;
   Do[m1 = m1 + If[k ≤ m, k, m], {k, n}];
   model = Array[mm, {n, m1 + Floor[n / 2]}] /. mm[i_, j_] → 1;
        n1 = 0;
   Do[     Do[ If[k == 1, (model[[j, n1 + 1]] = H; n1 = n1 + 1), If[k ≤ m,
            (model[[j, n1 + 1]] = Rk; model[[j + k - 1, n1 + 1]] = C;
        n1 = n1 + 1)]
           ],  {k, n - j + 1}],
       {j, n}];
   Do[ model[[j, n1 + j]] = SW;
    model[[n - j + 1, n1 + j]] = SW , {j, Floor[n / 2]}];
   model ]
```

In case of 5 qubits and $m = 3$, for example, we have

```
In[52]:= modelFourierApp[5, 3] // MatrixForm
```

Out[52]//MatrixForm=
$$\begin{pmatrix} H & R_2 & R_3 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & SW & 1 \\ 1 & C & 1 & H & R_2 & R_3 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & SW \\ 1 & 1 & C & 1 & C & 1 & H & R_2 & R_3 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & C & 1 & C & 1 & H & R_2 & 1 & 1 & SW \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & C & 1 & C & H & SW & 1 \end{pmatrix}$$

```
In[53]:= circuit[modelFourierApp[5, 3], {x4, x3, x2, x1, x0}, {, , , ,}]
```

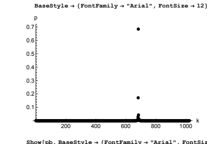
Out[53]=


What will happen if the AQFT is used instead of the full QFT in the algorithm for phase estimation?

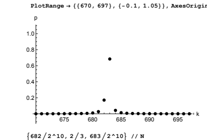
```
vec3a = Transpose[Conjugate[matrixU[modelFourierApp[10, 9] ]]].vec2 // Normal ;
vec3a /. φ → 17 / 32 // N // Chop
Position[%, 1.]
```

{{545}}

```
vec4a = N[Abs[#]^2] & /@ (vec3a /. φ → 2 / 3) // Chop;
dat1 = Table[{i - 1, vec4a[[i]]}, {i, Length[vec4a]}];
pb = ListPlot[dat1, PlotRange → All, PlotStyle → {Black, PointSize[0.02]},
  BaseStyle → {FontFamily → "Arial", FontSize → 12}, AxesLabel → {"k", "p"}]
```



```
Show[pb, BaseStyle → {FontFamily → "Arial", FontSize → 12}, AxesLabel → {"k", "p"},
 PlotRange → {{670, 697}, {-0.1, 1.05}}, AxesOrigin → {670, 0}]
```



```
{682 / 2^10, 2 / 3, 683 / 2^10} // N
```

{0.666016, 0.666667, 0.666992}

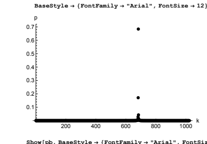
```
{dat1[[683, 2]], dat1[[684, 2]]}
```

{0.17098, 0.683916}

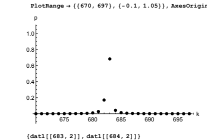
```
vec3b = Transpose[Conjugate[matrixU[modelFourierApp[10, 8] ]]].vec2 // Normal ;
vec3b /. φ → 17 / 32 // N // Chop
Position[%, 1.]
```

{{545}}

```
vec4b = N[Abs[#]^2] & /@ (vec3b /. φ → 2 / 3);
dat1 = Table[{i - 1, vec4b[[i]]}, {i, Length[vec4b]}];
pb = ListPlot[dat1, PlotRange → All, PlotStyle → {Black, PointSize[0.02]},
  BaseStyle → {FontFamily → "Arial", FontSize → 12}, AxesLabel → {"k", "p"}]
```



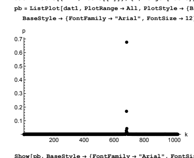
```
Show[pb, BaseStyle → {FontFamily → "Arial", FontSize → 12}, AxesLabel → {"k", "p"},
 PlotRange → {{670, 697}, {-0.1, 1.05}}, AxesOrigin → {670, 0}]
```



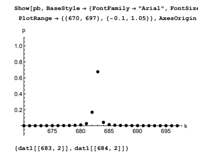
```
{dat1[[683, 2]], dat1[[684, 2]]}
```

{0.170969, 0.683865}

```
vec3c = Transpose[Conjugate[matrixU[modelFourierApp[10, 5] ]]].vec2 // Normal ;
vec3c /. φ → 17 / 32 // N // Chop
vec4c = N[Abs[#]^2] & /@ (vec3c /. φ → 2 / 3) // Chop;
dat1 = Table[{i - 1, vec4c[[i]]}, {i, Length[vec4c]}];
pb = ListPlot[dat1, PlotRange → All, PlotStyle → {Black, PointSize[0.02]},
  BaseStyle → {FontFamily → "Arial", FontSize → 12}, AxesLabel → {"k", "p"}]
```



```
Show[pb, BaseStyle → {FontFamily → "Arial", FontSize → 12}, AxesLabel → {"k", "p"},
 PlotRange → {{670, 697}, {-0.1, 1.05}}, AxesOrigin → {670, 0}]
```



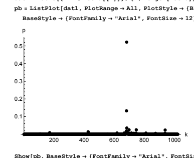
```
{dat1[[683, 2]], dat1[[684, 2]]}
```

{0.169218, 0.676145}

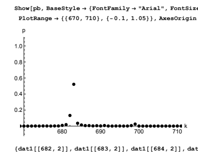
```
dat1[[683, 2]] + dat1[[684, 2]]
```

0.845363

```
vec3d = Transpose[Conjugate[matrixU[modelFourierApp[10, 3] ]]].vec2 // Normal ;
vec3d /. φ → 17 / 32 // N // Chop
vec4d = N[Abs[#]^2] & /@ (vec3d /. φ → 2 / 3) // Chop;
dat1 = Table[{i - 1, vec4d[[i]]}, {i, Length[vec4d]}];
pb = ListPlot[dat1, PlotRange → All, PlotStyle → {Black, PointSize[0.02]},
  BaseStyle → {FontFamily → "Arial", FontSize → 12}, AxesLabel → {"k", "p"}]
```



```
Show[pb, BaseStyle → {FontFamily → "Arial", FontSize → 12}, AxesLabel → {"k", "p"},
 PlotRange → {{670, 710}, {-0.1, 1.05}}, AxesOrigin → {670, 0}]
```



```
{dat1[[682, 2]], dat1[[683, 2]], dat1[[684, 2]], dat1[[685, 2]]}
```

{0.0182946, 0.132648, 0.521551, 0.0355429}

```
dat1[[683, 2]] + dat1[[684, 2]]
```

0.654199

According to Barenco, probability to obtain an accurate output using AQFT with at most *m* phase shift operations, when $m \ge \log_2 n + 2$, s at least
$P \ge \frac{8}{\pi^2} \sin^2\left(\frac{\pi}{4} \frac{m}{n}\right)$.

Remind that in case of the full QFT the lower bound for this probability is $\frac{8}{\pi^2}$. Another result was given by Cheung [D. Cheung, Improved Bounds for the Approximate QFT, arXiv: abs/quant-ph/0403071, 2004]
$P \ge \frac{4}{\pi^2} - \frac{1}{4n}$.

Our estimation is given by

```
In[55]:= n = 10;
m = 5;
Cos[π / 2^m]^(2 n - 2 m) // N
```

0.952877

```
Table[8/π^2 Sin[π m1 / 4 / n]^2 // N, {m1, m, n}]
```

{0.118705, 0.167064, 0.221289, 0.280045, 0.341884, 0.405285}

```
4/π^2 - 1/(4 n) // N
```

0.380285

References

Shor P.W. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. SIAM J. Comp. 26(5) (1997) 1484–1509.

Grover L.K. Quantum mechanics helps in searching for a needle in a haystack. Phys. Rev. Lett. 79 (1997) 325–328.

Nielsen, M., Chuang, I.: Quantum Computation and Quantum Information, 10th ed., Cambridge University Press, 2010.

http://www.quantiki.org/wiki/List_of_QC_simulators

Gerdt V.P., Kragler R., Prokopenya A.N. A Mathematica package for simulation of quantum computation. In: "Computer Algebra in Scientific Computing / CASC'2009", V. P. Gerdt, E. W. Mayr, E. V. Vorozhtsov (Eds.), LNCS 5743, Springer-Verlag, Berlin (2009) 106–117.

Gerdt, V.P. and Prokopenya, A.N.: Some algorithms for calculating unitary matrices for quantum circuits. Programming and Computer Software, vol. 36, No. 2 (2010) 111–116.

Gerdt V.P., Prokopenya A.N. The circuit model of quantum computation and its simulation with Mathematica. In: "Mathematical Modeling and Computational Science", G. Adam, J. Buša, M. Hnatič (Eds.), LNCS 7125, Springer-Verlag, Berlin Heidelberg (2012) 43–55.

Gerdt, V.P. and Prokopenya, A.N.: Simulation of quantum error correction my means of QuantumCircuit package. Programming and Computer Software, vol. 39, No. 3 (2013) 143–149.

Gerdt V.P., Prokopenya A.N. Simulation of quantum error correction with Mathematica. In: "Computer Algebra in Scientific Computing / CASC'2013", V. P. Gerdt, W. Koepf, E. W. Mayr, E. V. Vorozhtsov (Eds.), LNCS 8136, Springer-Verlag, Berlin (2013) 116-129.