Pakiet Quantum Computing Simulator w środowisku Python

Marek Sawerwain

26 czerwca 2008

Streszczenie

Obliczenia kwantowe to nowa dziedzina informatyki wprowadzająca inne zasady przetwarzania informacji. Pomimo postępu jaki dokonuje się na poziomie teorii związanej z obliczeniami kwantowymi, to technologia związana z obliczeniami tego typu nadal nie wychodzi poza ramy laboratoriów. Dlatego, nadal bardzo istotne są symulatory obliczeń kwantowych które pozwalają na zapoznanie się ze wszystkimi najważniejszymi aspektami, choć nie pozwalają one na przeprowadzanie symulacji obliczeń zawierających wiele qubitów. W artykule zostanie przedstawiony jeden z kilku dostępnych pakietów przeznaczonych do realizacji obliczeń kwantowych. Zostanie zaprezentowany sposób użycia tytułowego pakietu w środowisku Python. Zaprezentowane zostaną także ogólnie znane algorytmy kwantowe oraz sposób ich implementacji w ramach omawianego środowiska.

Spis treści

1 Wprowadzenie

Pakiet **QCS** jest zestawem funkcji opracowanych w języku C umożliwiającym przeprowadzanie symulacji obliczeń kwantowych. Dlatego, aby ułatwić wykorzystanie API oferowane przez **QCS** , został stworzony specjalny port dla języka Python (istnieje też port biblioteki **QCS** dla języka Java).

Podstawową zaletą opisywanego systemu (prace na pakietem rozpoczęły się w latach 2004 oraz 2005 [10]), który został oznaczony skrótem **QCS** (ang. Quantum Computing Simulator) jest niespotykana w innych tego typu programach [11, 12, 13] dostępność kilku podstawowych typów obwodów kwantowych. W pakiecie **QCS** możliwa jest symulacja obwodów typu PQC, CHP jak również symulacja pełnego modelu obwodowego dla qubitów oraz dla quditów.

Wewnętrzna architektura **QCS** pozwala, nie tylko na przeprowadzanie symulacji, lecz może się stać podstawą języka programowania kwantowego lub innego dowolnego systemu przystosowanego do szczególnych potrzeb użytkownika. Jest to szczególnie ważny aspekt, ponieważ inne tego typu systemy zazwyczaj są zamkniętymi środowiskami, które trudno rozszerzać i modyfikować.

Symulator obliczeń kwantowych **QCS** jest projektem ogólnego przeznaczenia. Został opracowany jako biblioteka funkcji zaimplementowana w języku ANSI C, przy wykorzystaniu bibliotek funkcji numerycznych BLAS i LAPACK. Ponieważ dostępna jest wersja tych bibliotek dla systemów równoległych, to istnieje możliwość przeniesienia pakietu **QCS** na maszyny równoległe.

Symulator pozwala na tworzenia skryptów w kilkunastu różnych językach programowania (przede wszystkim jest to wspomniany na samym początku Python czy Java). Z powodu wymienionych uwag oraz mając na uwadze dostępność pakietu i jego rolę edukacyjną, pakiet **QCS** powstał na bazie darmowego oprogramowania. Zostały zastosowane min. następujące pakiety:

- kompilator GCC v4.x
- system skryptowy SWIG v1.3.xx
- język Python v2.4
- pakiety do algebry liniowej LAPACK, BLAS, ATLAS
- pakiet do obliczeń równoległych MPICH-G2 (Win32, Linux)

Wszystkie wymienione narzędzia są oprogramowaniem darmowym opartym o licencję typu GNU GPL bądź podobną. Umożliwia to obniżenie kosztów tworzenia oprogramowania, jednakże nie ogranicza w żadnym razie funkcjonalności oprogramowania. Zastosowanie przede wszystkim kompilatora GCC, pozwala na uzyskanie bardzo istotnej własności, a mianowicie przenośności oprogramowania pomiędzy różnymi architekturami sprzętowo-programowymi.

Oprogramowanie na obecną chwilę jest dostępne dla najpopularniejszych systemów jak środowisko Windows w wersjach 32 i 64 bitowych oraz analogicznie system Linux. Oprogramowanie było także testowane na platformie IA-64 Itanium II.

Opisany system symulacji kwantowego modelu obliczeń był już stosowany podczas zajęć w ramach przedmiotu "Wstęp do informatyki kwantowej" na czwartym roku studiów magisterskich na kierunku informatyka na Uniwersytecie Zielonogórskim.

System był wykorzystywany przez studentów do realizacji różnych zadań. Zadania te dotyczyły min.: badania algorytmu Grovera, badania jakości kopiowania stanów kwantowych oraz opracowywania obwodów kwantowych realizujących algorytm Shora. Należy podkreślić interesujący aspekt opisywanego pakietu, a mianowicie zastosowania tego systemu do badania wydajności systemu komputerowego. Zarówno pod względem wydajności obliczeniowej procesora oraz wydajności operacji na pamięci.

System **QCS** podlega także ciągłemu rozwojowi, aktualnie ulepszeniu podlega moduł do równoległych symulacji obliczeń kwantowych. Nowym elementem jest możliwość przeprowadzania symulacji w ramach modelu jednokierunkowego. Jak dotąd, nie istniało oprogramowanie wspomagające symulacje tego rodzaju.

2 Obliczenia kwantowe

[do uzupełnienia]

3 Praca interaktywna z interpreterem języka Python

3.1 Podstawowe instrukcje sterujące w języku Python

Python to łatwy do opanowania język programowania, jednakże łatwość opanowania języka nie oznacza iż jest to język słaby. Wspiera struktury danych wysokiego poziomu oraz oferuje model programowania obiektowego. Zaletą języka jest bardzo spójna syntaktyka której nie sposób odmówić swoistej elegancji. Jest to także język interpretowany co powoduje iż znakomicie nadaje się do zastosowań skryptów oraz do szybkiego prototypowania.

3.2 Jak z Python'a zrobić kalkulator?

Ponieważ Python to język interpretowany, więc możliwa jest z nim praca interaktywna polegająca na tym, że użytkownik podaje polecenia a Python natychmiast je wykonuje i wyświetla wyniki na ekranie. Dobrym przykładem jest obliczenie wartość sumy od 1 do 10. Należy wpisać do interpretera następujące wyrażenie i nacisnąć [Enter] nakazując w ten sposób wykonanie polecenia (znaku >>> nie wpisujemy, jest to znak zachęty wyświetlany przez interpreter):

```
>>> 1+2+3+4+5+6+7+8+9+10
55
```

Środowisko odpowie natychmiast i wyświetli na ekranie wynik, czyli 55. Możemy wpisywać dowolne wyrażenia arytmetyczne posiłkując się, jeśli trzeba, nawiasami okrągłymi. Język oferuje mam wiele różnych operatorów. Wszystkie operatory, nie tylko arytmetyczne, wraz z ich znaczeniem i przykładem zastosowania zostały zebrane w tabeli 1.

W dalszej części tego artykułu została przyjęta następująco konwencja. Bowiem, jeśli linia rozpoczyna się od trzech znaków większości, oznacza to, że to my musimy wpisać odpowiednie wyrażenie. Jest tak również, gdy linia rozpoczyna się od trzech kropek. Natomiast linie nie rozpoczynające się od wymienionych znaków oznaczają odpowiedź Pythona. W przykładzie powyżej liczba 55 nie jest poprzedzona dodatkowymi znakami, czyli jest to wynik działania, który sam pojawi się na ekranie.

Wykorzystajmy fakt, że mamy dostępny operator potęgowania ** i obliczmy ile to jest 2 do potęgi 32 oraz 2 do 64:

```
>>>  2 ** 32
4294967296L
>>> 2 ** 64
18446744073709551616L
```

Są to spore liczby, szczególnie 2 do 64, ale Python bez problemów obliczy nawet 2 do 128 czy 512. Inne języki komputerowe, np.: C czy Pascal, nie oferują bezpośrednio możliwości działania na tak dużych liczbach. Ich obsługa jest jedną z wielu zalet Pythona. Zwróćmy uwagę, że duże liczby są oznaczane przez system dużą literą L na końcu.

3.3 Zmienne

Zmienna jest niczym worek, w których możemy przechowywać różne rzeczy. Raz może to być liczba a innym razem cały zbiór liczb bądź jakiś ciąg liter i znaków. Ponieważ komputer musi w jakiś sposób rozróżniać zmienne, to nadajemy im nazwy. Mogą to być pojedyncze litery, jednak lepiej, aby nazwa zmiennej była charakterystyczna np.: pole_kwadratu. W nazwach nie należy stosować polskich liter, trzeba także pamiętać, że Python rozróżnia małe i duże litery (ala i Ala to dla Pythona dwie różne zmienne). Podając następujące wyrażenie:

```
>>> a=3
```

Python utworzy zmienną o nazwie a, a jej wartością będzie trzy. Jeśli teraz wpiszemy samo a i naciśniemy [Enter], Python wyświetli na ekranie zawartość tej zmiennej. Gdyby ktoś przez pomyłkę podał dużą literę A, to zobaczy komunikat o błędzie, informujący, że zmienna A nie jest zdefiniowana. Można też powiedzieć, że zmienna nie istnieje.

```
>>> A
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in ?
    A
NameError: name 'A' is not defined
```

Kilka zmiennych można definiować w jednej linii:

```
>>> a=3; b=3; c=10;
```

W powyższym przykładzie znak równości oznacza tzw. instrukcję przypisania. Działanie tej instrukcji polega na nadaniu zmiennej nowej wartości. Każde wyrażenie przypisania jest zakończone średnikiem (gdybyśmy umieścili w jednej linii tylko jedno przypisanie, średnik nie byłby potrzebny). Przy nadawaniu wartości zmiennej wolno wykorzystywać inne zmienne. Utwórzmy zmienną o nazwie iloczyn i przypiszmy jej wartość iloczynu trzech zmiennych a, b, c:

```
>>> iloczyn=a*b*c
```

Możemy sprawdzić (wpisując nazwę zmiennej: iloczyn), że Python prawidłowo wykonał mnożenie i pamięta jego wynik w nowo stworzonej zmiennej. Na tym etapie niezwykle istotne jest zrozumienie, że znak = jest symbolem przypisania, a nie równości! Wyrażenia a=3 nie czytamy: a jest równe trzy, ale: niech od teraz a będzie równe trzy. Widać to szczególnie wyraźnie w takim (szokującym niekiedy matematyków) przykładzie:

```
>>>a=a+1
```

Powyższa instrukcja oznacza: niech od teraz a będzie równe swojej dotychczasowej zawartości powiększonej o jeden i jest bardzo często stosowana w praktyce. Sprawdźmy, jak to działa:

```
>>> a=0
>>> a=a+1
>>> print a
1
>>> a=a+1
>>> print a
2
```

Python posiada jedną specjalną zmienną, oznaczaną znakiem podkreślenia – „_”. W tej zmiennej zawarty jest zawsze wynik ostatniego wyrażenia. Zmienna ta może uczestniczyć w obliczeniach np.:

```
>>> 1.0/2.0 + 2.0/3.0 + 3.0/4.0
1.9166666666666665
>>> _ * 2
3.833333333333333
```

3.4 Pułapki, pułapki ...

Tworzenie programów wymaga znajomości subtelnych szczegółów. Policzmy sumę dwóch ułamków zwykłych. Ułamki zapiszemy przy pomocy operatora /, czyli znaku dzielenia (bo kreska ułamkowa to inaczej znak dzielenia):

```
>>> 1/2 + 1/2
```

Odpowiedź komputera może wprawić w zdumienie, ponieważ odpowie nam: zero. Python analizując wyrażenie 1/2 stwierdza, że obydwie liczby są całkowite, toteż stosuje całkowite dzielenie - z wynikiem zaokrąglonym w dół. A nie było to naszą intencją. Najbardziej poprawnym postępowaniem będzie, jeśli dopiszemy znak kropki oraz zero do każdej z liczb. W takim przypadku Python rozpozna liczy rzeczywiste i zastosuje odpowiednie dzielenie:

```
>>> 1.0/2.0 + 1.0/2.0
```

Pozostańmy przy tego typu problemach i sprawdźmy, czy dla trzech liczb uda się nam zbudować trójkąt prostokątny. Wykorzystamy do tego twierdzenie Pitagorasa. Poszczególne długości boków a, b, c będziemy przechowywać w zmiennych.

3.5 Mały przerywnik o modułach

Bezpośrednio po uruchomieniu interpretera Pythona nie mamy dostępu do wielu funkcji. Należy wczytać (zaimportować) odpowiedni moduł zawierający potrzebne funkcje.

Języki programowania zawierają bardzo niewiele komend - najczęściej nie więcej niż kilkanaście. Aby ułatwić pisanie w nich programów, tworzy się zestawy gotowych klocków - programików spełniających pożyteczne funkcje i w ten sposób ułatwiających pracę programiście. W Pythonie takie zbiory funkcji nazywają się modułami. Dostępnych jest wiele modułów, służą do rozmaitych zadań: tworzenia efektów graficznych, muzycznych, wyświetlania okienek; bardziej zaawansowani programiści mogą tworzyć własne.

Python zawiera standardowo wiele gotowych modułów. Ich zbiór nazywa się biblioteką standardową. Jednym z modułów pochodzących ze standardowej biblioteki jest moduł o nazwie math, który - jak łatwo zgadnąć - zawiera wiele przydatnych funkcji matematycznych (dokładny opis można odszukać w dokumentacji rozdział Library Reference, a w polskiej wersji dokumentacji Opis biblioteki standardowej). Zobaczmy, jak korzysta się z modułów. Przede wszystkim należy poinformować komputer, których modułów chcemy używać. Dokonujemy tego słowem kluczowym import. Na przykład, jeśli chcemy korzystać z funkcji modułu math wykonujemy komendę:

```
>>> import math
```

Jeśli chcemy skorzystać z jakiejkolwiek funkcji modułu (dość często mówi się o wywoływaniu funkcji), musimy poinformować komputer, w którym module powinien jej szukać (jednocześnie możemy mieć zaimportowanych wiele różnych modułów). Fachowo nazywa się to podaniem nazwy przestrzeni, w jakiej się znajduje używana przez nas funkcja. Brzmi groźnie, ale polega tylko na tym, że gdy obliczamy np. pierwiastek kwadratowy, a funkcja, która oblicza taki pierwiastek, nosi nazwę sqrt, to dodatkowo poprzedzamy ją nazwą modułu (rozdzielając nazwę modułu i funkcji znakiem kropki), z którego pochodzi:

```
>>> math.sqrt(9)
3
```

Argument dowolnej funkcji obejmujemy nawiasami a jeśli argumentów jest kilka, to oddzielamy je przecinkami. Wykorzystajmy teraz naszą wiedzę do obliczenia długości przeciwprostokątnej dla dwóch boków o długości odpowiednio 4 i 2. Wpisujemy po kolei:

```
>>> import math;
>>> a=2
>>> b=4
>>> math.sqrt(a*a+b*b)
4.4721359549995796
```

W tym prostym przykładzie zostały użyte dwie zmienne, ale nie ma żadnych przeciwwskazań, aby z nich zrezygnować i jako argument funkcji sqrt podać sumę kwadratów bezpośrednio.

Powróćmy do naszego trójkąta prostokątnego. Sprawdzenie czy z boków o długościach zapisanych w zmiennych a, b, c można zbudować trójkąt wykonany używając operatora porównania, który składa się z dwóch znaków równości – == (dzięki czemu wiadomo, że chodzi o porównanie a nie o przypisanie). Sprawdzenie wygląda następująco:

```
>>>c**2 == a**2 + b**2
```

3.6 Pułapek ciąg dalszy

Sprawdzamy, czy kwadrat przeciwprostokątnej (zmienna **c**) jest równy sumie kwadratów boków **a** i **b**. Python odpowie wartością jeden, jeśli podana równość jest prawdziwa, lub zerem, gdy nasz test zawiódł. Trzeba pamiętać, że wartość jeden oznacza logiczną prawdę, a wartość zero to logiczny fałsz.

Choć maszyny są zdolne przeprowadzać ogromne ilości obliczeń w ułamkach sekund, to niestety, często powstają błędy związane z zaokrąglaniem wyników. Świadczy o tym taki najprostszy przykład, mamy następujące zmienne reprezentujące boki trójkąta:

```
>>> a=2
>>> b=2
```

Obliczamy długość przeciwprostokątnej:

```
>>> c=math.sqrt(a*a+b*b)
```

Sprawdzamy, czy dane spełniają twierdzenie Pitagorasa:

```
>>> c*c==a*a+b*b
```

Każdy ze zdziwieniem stwierdzi, że nie, bo w odpowiedzi uzyskaliśmy wartość zero. Dlatego tak jest? Wystarczy sprawdzić ile wynosi kwadrat zmiennej **c**:

```
>>> c*c
8.0000000000000018
```

Natomiast suma kwadratów:

```
>>> a*a+b*b
8
```

Te dwie liczby, jak widać, się różnią, choć dopiero na szesnastym miejscu po przecinku (o ile dobrze policzone zostały zera ;-)). Różnica praktycznie żadna, ale komputer wykryje taką różnicę. Jest to niestety błąd, który może bardzo poważnie wpłynąć na dalsze obliczenia doprowadzając ostateczne wyniki do absurdu.

3.7 Instrukcje sterujące

Do podstawowych instrukcji sterujących należą: instrukcja warunkowa **if**, instrukcje pętli **for** oraz **while**. Wymienione trzy instrukcje w odniesieniu do innych języków programowania różnią się zasadniczo sposobem zapisu, choć pętla for w przypadku Pythona ma np.: w stosunku do pętli for w języku C mniejsze możliwości. Możliwe jest także tworzenie funkcji. Jednakże oprócz wymienionych klasycznych struktur sterujących język Python wspiera także pewne elementy programowania funkcjonalnego jak wyrażenia lambda oraz funkcja **map**.

3.7.1 Instrukcja warunkowa if

Sposób funkcjonowania instrukcji warunkowej jest naturalnie typowy dla tego typu instrukcji. Jeśli warunek jest prawdziwy to wykonywane są instrukcje zawarte w bloku instrukcji warunkowej. Przy czym nie ma jak np.: w języku Pascal słów kluczowych `begin` oraz `end` bądź nawiasów `{` i `}` znanych choćby z języka C. W języku Python blok instrukcji oznacza się za pomocą wcięć wykonywanych przy pomocy spacji lub klawisza TAB. Należy jednak tworząc blok instrukcji za pomocą wcięć zachować konsekwencję, jeden blok może zostać utworzony za pomocą klawisza TAB albo za pomocą spacji:

```
if warunek:
    instrukcja 1
    instrukcja 2
    instrukcja 3
```

Instrukcja **if** posiada także blok **else** oraz blok **elif** w którym określa się warunek do sprawdzenia jeśli warunek nadrzędny zawiedzie. Poniższy przykład powinien być wprowadzony z konsoli, trzy kropki oznaczają symbol zachęty do wprowadzania dalszej części wyrażenia (funkcja **int** wykonuje konwersję na typ całkowity, natomiast **raw_input** pozwala na odbiór danych z klawiatury):

```
>>> x = int(raw_input("Proszę podać liczbę "))
>>> if x < 0:
...      x = 0
...      print 'Liczba ujemna'
... elif x == 0:
```

```
...      print 'Zero'
... elif x == 1:
...      print 'Jedynka'
... else:
...      print 'Więcej niż jedynka'
...
```

3.7.2 Pętla typu for

Pętla for w Pythonie ma nieco inną składnię niż w językach C oraz C++, ponieważ zakres elementów jakie może przyjmować główna zmienna pętli pętla nie jest tylko i wyłącznie ograniczony do typów prostych porządkowych, czego przykładem mogą być liczby całkowite. W przykładzie obliczamy długości zadanych ciągów danych zapisanych w postaci listy. Poszczególne elementy listy będą kolejno przepisywane do zmiennej **x**. A w treści pętli za pomocą polecenia **print** wyświetlimy dany wyraz oraz jego długość:

```
>>> a = ['kot', 'okno', 'książka']
>>> for x in a:
...     print x, len(x)
...
kot 3
okno 4
książka 12
```

3.7.3 Pętla typu while

Pętla **while**, czyli pętla typu „dopóki" działa w sposób identyczny jak tego typu pętle w innych języka programowania. Pętla wykonuje swój blok instrukcji tak długo warunek pętli jest prawdziwy.

```
a=1
while a <= 10:
    print a*a
    a=a+1
```

3.8 Tworzenie nowych funkcji

Tworzenie nowych funkcji w Pythonie jest dość proste. Wystarczy bowiem nowy blok danych poprzedzić nagłówkiem rozpoczynającym się słowem **def**, następnie podajemy nazwę funkcji a w nawiasie ewentualne parametry.

```
>>> def fib(n):
...     a, b = 0, 1
...     while b < n:
...         print b,
...         a, b = b, a+b
...
>>> fib(2000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

3.9 Załadowanie modułu obliczeń kwantowych

Pierwsza czynność związana z użytkowaniem systemu **QCS** w języku *Python* to załadowanie modułu o nazwie **QCS** (język *Python* rozpoznaje duże i małe litery, próba załadowania modułu o nazwie np.: **QCS** niestety nie powiedzie się). Wczytanie modułu odbywa się za pomocą funkcji **import**, zarówno podczas pracy interaktywnej na poziomie konsoli oraz w trybie tworzenia skryptów, wydanie polecenia importu jest następujące:

```
>>> import qcs
```

Po załadowaniu modułu **qcs** w trybie konsoli wyświetli się nagłówek systemu symulacji obliczeń kwantowych, o następującej postaci (lub podobnej w zależności od wersji systemu QCS):

```
/---\ /---\ /---\
|   | |     |             Quantum
|   | |     \---\     Computing System
|   | |         |
\-\-/ \---/ \---/
   \
Quantum Computing System v0.2.18
Release name: cucumber
: compilation date (Apr  4 2008 21:14:01)
: compiled on Win32 by GCC 3.4.5 (mingw special) from MinGW v3.14
+ matrix set is already generated
```

Naturalnie dowolne inne moduły np.: pakiet funkcji matematycznych również wczytujemy za pomocą funkcji import.

4 Tworzenie rejestru kwantowego

Główną funkcją tworzącą rejestr w systemie **QCS** jest funckja **QuantumReg**. Z racji roli jaka ona pełni lepiej jednak w przypadku o konstruktorze rejestr kwantowego.

4.1 Rejestr zawierający qubity

Utworzenie rejestru o dwóch qubitach, wymaga wywołania funkcji **QuantumReg** z jednym argumentem oznaczającym liczbę qubitów jaka znajdzie się w rejestrze:

```
q = qcs.QuantumReg(2)
```

Utworzony rejestr jednak w pełni wyzerowany, o czym można się przekonać wyświetlając pełny opis stanu rejestry q za pomocą metody **PrFull**. Łatwo się o tym przekonać wykonując dwa proste polecenia, które utworzą rejestr a następnie wyświetlą jego zawartość:

```
>>> q=qcs.QuantumReg(2)
QuantumReg(int i) new 2 qubits
>>> q.PrFull()
0.000000 + 0.000000i |00>
0.000000 + 0.000000i |01>
0.000000 + 0.000000i |10>
0.000000 + 0.000000i |11>
>>>
```

Na rejestrze wypełnionym zerami, niestety nie będzie można wykonać żadnej operacji. Koniecznie jest przeprowadzenie operacji resetowania, czyli wprowadzenia go w stan $|00\rangle$. Operacja ta jest realizowana przez metodę **Reset**.

```
>>> q.Reset()
>>> q.PrFull()
1.000000 + 0.000000i |00>
0.000000 + 0.000000i |01>
0.000000 + 0.000000i |10>
0.000000 + 0.000000i |11>
>>>
```

Tym razem rejestr znajduje się w stanie $|00\rangle$, amplituda prawdopodobieństwa dla tego stanu jest równa zero, pozostałe stany jak widać nie występują, ich amplitudy są równe zero.

Rejestr można wprowadzić w dowolny stan bazowy wykorzystując metodę **SetKet**. W jej argumencie należy podać wielkość binarną która określi interesujący nas stan np.: `q.SetKet("01")`, oznacza iż rejestr zostanie wprowadzony stan gdzie pierwszy qubit znajdzie się w stanie zero natomiast drugi w stanie jeden. Podobnie `q.SetKet("11")` ale tym razem obydwa qubity znajdą się w stanie jeden. Bezpośrednim odpowiednikiem metody **Reset** jest polecenie `q.SetKet("00")`.

4.2 Rejestr zawierający qudity

Utworzenie rejestru zawierającego dwa qudity, o czterech stopniach swobody wymaga podanie naturalnie dwóch argumentów dla konstruktora QuantumReg, pierwszy to liczba quditów, natomiast drugi parametr określa stopień swobody, czyli liczbę stanów bazowych.

```
>>> q = qcs.QuantumReg(2,4)
```

Sposób korzystania z takiego rejestru jest identyczny jak z rejestru qubitowego. Jednakże nie wszystkie dostępne bramki jednoqubitowe dostępne w ramach pakietu **QCS** funkcjonują poprawnie dla rejestru zawierającego qudity[1].

4.3 Wyświetlanie stanu rejestru

Podstawową metodą wyświetlającą stan rejestru jest już wspomniana metoda **Pr**. Istnieją dwie dodatkowe metody **PrBin** oraz **PrDec**, które analogicznie jak metoda Pr wyświetlają niezerowe amplitudy. Jednakże pierwsza z nich wyświetla numer stanu zawsze w postaci binarnej, natomiast druga PrDec zawsze w postaci liczby dziesiętnej. Jest to istotne w przypadku quditów, metoda PrBin wyświetla opis w postaci liczb w systemie o podstawie d, gdzie d to stopień swobody quditu. Natomiast metoda PrDec zamienia liczby systemu o podstawie d na liczby dziesiętne.

```
>>> q.Pr()
0.707107 + 0.000000i |00>
0.707107 + 0.000000i |11>
>>>
```

Istnieje też metoda o nazwie **PrSqr** a wyświetla ona kwadraty amplitud:

```
>>> q.PrSqr()
0.500000 |00>
0.500000 |11>
>>>
```

Działanie metody **PrFull** jest bardzo podobne do poprzednich, jedyną różnicą jest wyświetlanie wszystkich amplitud:

```
>>> q.PrFull()
0.707107 + 0.000000i |00>
0.000000 + 0.000000i |01>
0.000000 + 0.000000i |10>
0.707107 + 0.000000i |11>
>>>
```

Metody o nazwach **PrBinDec** oraz **PrBinDecSqr** pokazują stan rejestru z pominięciem amplitud o wartości zero. Druga metoda w odróżnieniu od pierwszej pokazuje kwadraty amplitud.

`>>> q.PrBinDec()`	`>>> q.PrBinDecSqr()`
`0.707107 + 0.000000i |00> |0>`	`0.500000 |00> |0>`
`0.707107 + 0.000000i |11> |3>`	`0.500000 |11> |3>`

[1]Nie oznacza to, że wywołanie bramki która nie posiada implementacji dla quditu modyfikuje stan rejestru kwantowego. System ignoruje wykonanie takiej operacji, inaczej mówiąc wykonywana jest operacja **noop** (ang. no operation – noop)

Na podstawie aktualnego stanu można także zobaczyć jak wygląda macierz gęstości dla aktualnego stanu rejestru – metoda **PrDenMat**:

```
>>> q.PrDenMat()
0.500000 0.000000 0.000000 0.500000
0.000000 0.000000 0.000000 0.000000
0.000000 0.000000 0.000000 0.000000
0.500000 0.000000 0.000000 0.500000
>>>
```

4.4 Inicjalizacja za pomocą tablicy qubitów

Wykonując odpowiednie operacje na rejestrze możemy przygotować potrzebny stan do dalszej pracy. Innym sposobem inicjalizacji jest przygotowanie tablicy qubitów. Na podstawie stanów poszczególnych qubitów, tworzony jest stan rejestry, czyli jest to superpozycja poszczególnych qubitów znajdujących się w tablicy. Posługiwanie się tablicą jest dość proste, dla istniejącego rejestru np.: o dwóch qubitach

```
q=qcs.QubitReg(2)
q.Reset()
```

należy utworzyć odpowiednią tablicę również zawierającą dwa qubity. Za utworzenie tablicy odpowiedzialny jest konstruktor **QubitArray**. Za argument przyjmuje on liczbę całkowitą, która naturalnie oznacza liczbę qubitów znajdujących się w tablicy.

```
qt=qcs.QubitArray(2)
```

Powstała tablica nie zawiera poprawnie zainicjalizowanych qubitów, toteż należy utworzyć dwa dodatkowe qubitu oraz dokonać ich inicjalizacji np.: w następujący sposób:

```
q0=qcs.Qubit() ; q0.SetThetaPsi(15, 0)
q1=qcs.Qubit() ; q1.SetThetaPsi(45, 0)
```

Zastosowana powyżej metoda **SetThetaPsi** jest dość wygodnym sposobem inicjalizacji, ponieważ podajemy tylko dwa parametry, a są wartości kątów w stopniach. Poszczególne qubitu należy jeszcze wstawić do tablicy co realizuje się za pomocą metody **SetQubitN**.

```
qt.SetQubitN(0, q0)
qt.SetQubitN(1, q1)
```

Sam proces inicjalizacji jest dość prosty i wymaga wywołania metody **SetFromQubitArray**, gdzie za argument wstawiamy tablicę z qubitami.

```
q.SetFromQubitArray(qt)
q.PrFull()
```

Stan rejestru po inicjalizacji jest następujący:

```
0.915976 + 0.000000i |00>
0.379410 + 0.000000i |01>
0.120590 + 0.000000i |10>
0.049950 + 0.000000i |11>
```

Istnieje też możliwość wyświetlenia wartości poszczególnych qubitów znajdujących się w tablicy. Odczyt qubitu znajdującego się w tablicy realizowany jest za pomocą funkcji **GetQubitN**. Stan pojedynczego qubitu podobnie jak rejestru można wyświetlić za pomocą metody **Pr**.

```
for i in [0, 1]:
    qt.GetQubitN(i).Pr()
```

Stan dwóch qubitów jest następujący:

```
(0.991445 + 0.000000i) |0> + (0.130526 + 0.000000i) |1>
(0.923880 + 0.000000i) |0> + (0.382683 + 0.000000i) |1>
```

4.5 Tablice quditów

W podobny sposób można postąpić w przypadku quditów. Jednakże ze względu na to iż qudit może posiadać wiele stopni swobody, toteż konieczne jest podczas tworzenia tabeli podanie dwóch argumentów. Pierwsza oznacza ilość quditów a drugi naturalnie stopień swobody:

```
qt=qcs.QuditArray(3,3)
```

Podczas tworzenie pojedynczych quditów konieczne jest naturalnie jest podanie dodatkowego argumentu oznaczającego ilość stanów bazowych:

```
q0=qcs.Qudit(3) ; q0.SetRandomState()
q1=qcs.Qudit(3) ; q1.SetRandomState()
q2=qcs.Qudit(3) ; q2.SetRandomState()
```

Użyta metoda **SetRandomState** nadaje losowy stan quditowi. Wstawienie quditu to zadanie dla funkcji **SetQuditN**, natomiast inicjalizacja rejestru wykonamy wywołując funkcję **SetFromQuditArray**.

5 Bramki kwantowe

W symulatorze **QCS** obowiązuje następujący sposób numerowania qubitów (jak również quditów). Qubit o numerze zero zawsze znajduje się po lewej stronie w zapisie stanu w postaci ketu:

$$|\psi\rangle = |0_0 0_1 0_2 0_3 0_4 0_6\rangle \tag{1}$$

5.1 Jednoqubitowe bramki

Podstawowe bramki Pauliego jest realizowane za pomocą następujących metod:

1. **PauliX** (inna nazwa to **NotN**)
2. **PauliY**
3. **PauliZ**

Bramki te są naturalnie bramkami jednoqubitowymi i przyjmują tylko jeden argument, liczbę całkowitą która jest numerem qubitu/quditu do którego chcemy zastosować daną bramkę. Następujący przykład pokazuje w jaki sposób używamy bramki Pauliego σ_Y. Jeśli wykonamy następujące polecenia:

```
q = qcs.QuantumReg(2)
q.Reset()
q.Pr()
q.PauliY(0)
q.Pr()
```

To dla stanu początkowy, który jest następujący:

```
1.000000 + 0.000000i |00>
```

zobaczymy, iż po użyciu metody PauliY na zerowym qubicie otrzymamy nowy stan:

```
0.000000 + 1.000000i |10>
```

Inną ważną bramką jest bramka Hadamarda reprezentowana przez metodę o nazwie **HadN**. Stosowanie tej bramki umożliwia tworzenie superpozycji. Dla qubitu znajdującego się w stanie $|0\rangle$ gdy zastosujemy bramkę Hadamadarda otrzymamy qubit znajdujący się w następującej superpozycji (pomijamy amplitudy prawdopodobieństwa): $|0\rangle + |1\rangle$. Sytuacja przedstawia się podobnie dla stanu $|1\rangle$ lecz wtedy otrzymujemy superpozycję o ujemny znaku: $|0\rangle - |1\rangle$

PauliX(n)	HadN(n)	RotAlphaN(n , alpha)	MYRot90N(n)
PauliY(n)	NotN(n)	RotThetaN (n , alpha)	MZRot90N(n)
PauliZ(n)	SquareRootN (n)	XRot90N(n)	CorrMatForTeleport(int n, int m)
GateT(n)	ArbitraryOneQubitGate(n, gate)	YRot90N(n)	
GateS(n)	Phase(n)	ZRot90N(n)	
GateV(n)	PhaseFN(n)	MXRot90N(n)	

Tabela 1: Spis dostępnych bramek jednoqubitowych: n jest liczbą całkowitą w zakresie $\langle 0, l-1\rangle$, gdzie l to liczba qubitów w rejestrze kwantowym, wartość *alpha* to kąt podany w radianach, natomiast *gate* to macierz stanowiąca jednoqubitową bramkę unitarną o wymiarach 2×2

Ważnym aspektem wynikającym z faktu iż bramka Hadamarda jest reprezentowana przez samosprzężony operator, to ponowne zastosowane bramki Hadamarda do wymienionych superpozycji powoduje, że powrócimy do stanu $|0\rangle$ bądź $|1\rangle$.

Pakiet **QCS** oferuje zbiór podstawowych bramek jednoqubitowych, które zostały wymienione w tabeli (1). Istotną rolę pełni metoda **ArbitraryOneQubitGate**. Pozwala ona na stosowanie dowolnej zdefiniowanej przez użytkownika bramki unitarnej w postaci podanej macierzy. Pierwszy argument to numer qubitu, do którego chcemy zastosować bramkę natomiast w drugim argumencie należy podać macierz reprezentującą bramkę unitarną.

```
q.ArbitraryOneQubitGate(0, mat)
```

Natomiast sam proces utworzenie macierzy np.: macierzy dla bramki NOT można zrealizować w następujący sposób:

```
m=qcs.Matrix(2,2)

m.AtDirect(0,1,1,0)
m.AtDirect(1,0,1,0)
m.Pr()
```

Metoda **AtDirect** pozwala na zmianę wartości macierzy w danym wierszu i kolumnie (dwa pierwsze argumenty). Natomiast w dwóch następnych argumentach wprowadzamy liczbę zespoloną, czyli wartość rzeczywistą i urojoną.

5.2 Bramki dwuqubitowe

Pakiet **QCS** oferuje wiele gotowych bramek działających na dwóch qubitach. Jeden qubit jest tzw. qubitem sterującym natomiast drugi qubit to cel gdzie następuje wykonanie wskazanej operacji. Podstawową bramką tego typu jest bramka **CNOT**, czyli bramka kontrolowanej negacji. Sposób jej funkcjonowania można określić w następujący sposób, jeśli qubit sterujacy znajduje się w stanie jeden ($|1\rangle$), to na drugim wskazanym qubicie zostanie wykonana w przypadku bramki **CNOT** operacja negacji.

Tabela 2 zawiera spis dostępnych bramek dwuqubitowych. Ogólna zasada działania tych bramek jest podobna przedstawionej przed chwilą bramki CNOT. Zaletą systemu QCS jest dowolność stosowania numerów qubitów w rejestrze dla poszczególnych bramek można napisać:

```
q.CNot(0,1)
```

jak również

```
q.CNot(1,0)
```

choć ta możliwość wydaje się całkowicie trywialna, to istnieją pakiety do symulacji obliczeń kwantowych, gdzie pierwszy argument musi być mniejszy niż drugi. Tabela 2 wymienia także bramki **CNot_zero** czy **CPauliX_zero**, są to bramki gdzie qubit sterujacy musi znajdować się w stanie zero aby nastąpiło wykonanie operacji na drugim stanie.

Zadbano także o wygodę, bramka **CRotAlpha** jest bramką stosowaną w odwrotnej kwantowej transformacie Fouriera, stosowanej w alg. Shora rozkładu liczby na czynniki pierwsze. Wartość parametru bramki określamy w trzecim argumencie:

```
q.CRotAlpha( 0, 1, 0.25 )
```

dwa pierwsze argumenty to naturalnie qubit kontrolujący oraz qubit kontrolowany.

Analogicznie jak dla bramek jednoqubitowych, istnieje możliwość tworzenia własnych bramek dwuqubitowych. Wykorzystujemy metodę o nazwie **Arbitrary2QubitGate**. Pierwsze dwa argumenty to numery qubitow, natomiast w trzecim argumencie podajemy macierz bramki, która zostanie zastosowana do qubitu kontrolowanego. Metoda ta wykonuje operację na qubit kontrolowanym określoną przez macierz gate_matrix, gdy qubit kontrolujący znajduje się w stanie $|1\rangle$.

```
q.Arbitrary2QubitGate(0,1, gate_matrix)
```

Natomiast metoda **Arbitrary2QubitGateZero** działa, gdy qubit kontrolujący znajduje się w stanie $|0\rangle$.

CNot(c, t)	CNot_zero(c, t)
CHad(c, t)	CHad_zero(c, t)
CPauliX(c, t)	CPauliX_zero(c, t)
CPauliY(c, t)	CPauliY_zero(c, t)
CPauliZ(c, t)	CPauliZ_zero(c, t)
CGateV(c, t)	CGateV_zero(c, t)
CGateS(c, t)	CGateS_zero(c, t)
CHad(c, t)	CHad_zero(c, t)
CPhaseF(c, t)	CPhaseF_zero(c, t)
CRot45(c, t)	CRot45_zero(c, t)
CRot90(c, t)	CRot90_zero(c, t)
CRotAlpha(c, t, alpha)	CRotAlpha_zero(c, t, alpha)
CRotTheta(c, t, theta)	CRotTheta_zero(c, t, theta)

Tabela 2: Spis dostępnych bramek dwuqubitowych: n jest liczbą całkowitą w zakresie $\langle 0, l-1\rangle$, gdzie l to liczba qubitów w rejestrze kwantowym, wartość *alpha* to kąt podany w radianach, natomiast *gate* to macierz stanowiąca bramkę unitarną o wymiarach 2×2

5.3 Bramki wieluqubitowe

Linii sterujących od których zależy wykonanie operacji na qubicie kontrolowanym naturalnie może być więcej. Maksymalna liczba jest ograniczona przez ilość qubitów w rejestrze. Przykładem tego typu bramkek jest tzw. bramka Toffoliego, czyli taka która posiada dwie linie sterujące oraz jedną kontrolowaną. W systemie **QCS** metoda odpowiadającą tej bramce metoda to CNot2(c1,c2,t). Łatwo też założyć iż jest jedna linia sterująca która dopuszcza wykonanie pewnej operacji na wielu qubitach. Pakiet **QCS** oferuje więcej funkcji podobnych do bramki Toffoliego, z tą różnicą iż liczba linii sterujących jest większa, a są to bramki CNot3, CNot4 oraz CNot5. Istnieją też odmiany tej bramki, gdzie wymienione linie sterujące są kontrolowane przez stan $|0\rangle$ np.: CHad3_zero. Pomocna dla bramek wieloqubitowych jest ponownie tabela 2. Większość bramek posiada odpowiedniki wieloqubitowe.

Inną bramką którą warto wymienić w gronie bramek wieloqubitowych to bramka typu **SWAP**. Dokonuje ona zamiany amplitud pomiędzy dwoma qubitami. Choć **QCS** nie oferuje wprost tej bramki to bardzo łatwo utworzyć taką bramkę za pomocą zwykłych bramek **CNot** w następujący sposób tworząc prostą funkcję w języku Python:

```
def swap(_r, a, b):
  _r.CNot(a, b)
  _r.CNot(b, a)
  _r.CNot(a, b)
```

6 Przykłady

6.1 Tworzenie par splątanych

Cztery podstawowe stany splątane nazywane stanami Bell'a lub parami EPR, posiadają następujące reprezentacje wektorowe:

$$\psi^+ = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ 0 \\ 0 \\ \frac{1}{\sqrt{2}} \end{bmatrix}, \quad \psi^- = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ 0 \\ 0 \\ -\frac{1}{\sqrt{2}} \end{bmatrix}, \quad \phi^+ = \begin{bmatrix} 0 \\ \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \\ 0 \end{bmatrix}, \quad \phi^- = \begin{bmatrix} 0 \\ \frac{1}{\sqrt{2}} \\ -\frac{1}{\sqrt{2}} \\ 0 \end{bmatrix}.$$

Wygodnie będzie przygotować cztery funkcje, które dla podanego rejestru o dwóch qubitach będą przygotowywać odpowiedni stan. W tym procesie bardzo przydatną rolę pełni funkcja **SetKet**, która przygotowuje wstępny stan.

```
def make_psi_plus(_r):
    _r.SetKet("00")
    _r.HadN(0)
    _r.CNot(0,1)

def make_psi_minus(_r):
    _r.SetKet("10")
    _r.HadN(0)
    _r.CNot(0,1)

def make_phi_plus(_r):
    _r.SetKet("10")
    _r.HadN(1)
    _r.CNot(1,0)

def make_phi_minus(_r):
    _r.SetKet("11")
    _r.HadN(0)
    _r.CNot(0,1)
```

Chcąc wyświetlić stan za pomocą metody **Pr**, wygodnie będzie wywoływać zdefiniowane powyżej funkcje:

```
r=qcs.QuantumReg(2)
r.Reset()

make_psi_plus(r) ; print "psi+" ; r.Pr()

make_psi_minus(r) ; print "psi-" ; r.Pr()

make_phi_plus(r) ; print "phi+" ; r.Pr()

make_phi_minus(r) ; print "phi-" ; r.Pr()
```

```
import qcs


_TELEPORT_QUBIT = 0
_ALICE_QUBIT = 1
_BOB_QUBIT = 2

q=qcs.QubitReg(3)
q.Reset()
q.SetKet("000")

q.HadN(_TELEPORT_QUBIT)

q.HadN(_ALICE_QUBIT)
q.CNot(_ALICE_QUBIT,
⟹ _BOB_QUBIT)
q.CNot(_TELEPORT_QUBIT,
⟹ _ALICE_QUBIT)
q.HadN(_TELEPORT_QUBIT)
```

```
v=q.MeasureN(_TELEPORT_QUBIT, _ALICE_QUBIT)
if v==0:
    q.Pr()
if v==1:
    q.NotN(_BOB_QUBIT)
    q.Pr()
if v==2:
    q.PhaseFN(_BOB_QUBIT)
    q.Pr()
if v==3:
    q.NotN(_BOB_QUBIT)
    q.PhaseFN(_BOB_QUBIT)
    q.Pr()

del q
```

Rysunek 1: Skrypt odpowiedzialny za symulację protokołu teleportacji dla dowolnego qubitu

```
import qcs
import math

def gcd(m,n):
    a=m
    b=n
    while b!=0:
        a,b = b, math.fmod(a,b)
    return a

def swap(r, a, b):
    r.CNot(a, b)
    r.CNot(b, a)
    r.CNot(a, b)

def PowerMod(r):
    r.CNot(2,4)
    r.CNot(2,5)
    r.CNot(3,5)
    r.CNot2(1,5,3)
    r.CNot(3,5)
    r.CNot(6,4)
    r.CNot2(1,4,6)
    r.CNot(6,4)
```

```
def iqft(r):
    r.HadN(0)
    r.CRot90(2,1)
    r.HadN(1)
    r.CRot45(2,0)
    r.CRot90(1,0)
    r.HadN(2)


x=7
n=15

r=qcs.QubitReg(7)
r.Reset()
r.SetKet("0000001")
r.HadN(0)
r.HadN(1)
r.HadN(2)
PowerMod(r)
m=r.MeasureN(3,6)
iqft(r)
swap(r,0,2)
y=r.MeasureN(0,2)
print "y=", y
```

```
i=gcd(y,8)
r=8/i
p1=gcd((x ** (r/2)) - 1, n)
p2=gcd((x ** (r/2)) + 1, n)

print "r=", r
print "p1=", p1
print "p2=", p2
print "p1*p2=", p1*p2

if p1*p2 != n:
    print "factorization failed!"
else:
    print "ok"
```

Rysunek 2: Skrypt implementujący algorythm Shora dla liczby 21

```
import qcs

q1=qcs.QuantumReg(4)
q1.Reset()
q1.SetKet("0001")
q1.Pr()

q=qcs.QuantumReg(4)
q.Reset()

q.Had()

m1=q.GroverChangeSignGen(1);
m2=q.GroverTurnAroundMean();
results=[]
print
for i in range(1,20):
    q.MatrixApply(m1)
    q.MatrixApply(m2)
    f=qcs.Fidelity(q1.GenDenMat(), q.GenDenMat())
    results=results+[f]

print "Fidelity "

for i in results:
i.Pr()
print
```

Rysunek 3: Skrypt implementujący algorytm Grovera na przykładzie wzmacniania amplitudy stany $|1\rangle$