Quantum Shell – emulator komputera kwantowego

Jakub Pilch

Cel projektu

Poniższa dokumentacja stanowi opis praktycznego podejścia do stworzenia emulatora komputera kwantowego. Wszystkie kroki opisane dalej mają ostatecznie utworzyć spójną koncepcję opisującą model komputera kwantowego możliwy do zasymulowania na standardowych maszynach opartych o procesory krzemowe.

Podstawowym celem było stworzenie abstrakcyjnego modelu komputera kwantowego możliwego do zasymulowania na klasycznym komputerze, a następnie utworzenie programu w języku C# na platformę .NET, który udostępni mechanizmy pozwalające obserwować działanie obliczeń kwantowych. Założeniem jest zaprojektowanie solidnej architektury programu, która pozwoli na jego rozbudowę (dodanie prostego asemblera dla bramek kwantowych, rozszerzanie o nowe algorytmy) w przyszłości.

Poniższy dokument jest drugim z dwóch stanowiących pełną dokumentację projektu Quantum Shell. Pierwszym z nich jest *Quantum Shell – obliczenia kwantowe* zawierający opis teoretyczny całego podejścia do zbudowania emulatora.

Strona projektu: https://github.com/3yakuya/Quantum-Shell

1. Architektura aplikacji

Jednym z założeń projektu było stworzenie jak najbardziej *uniwersalnego* emulatora komputera kwantowego, który nie powinien sprawiać trudności zarówno w użytkowaniu jak i rozszerzaniu jego funkcjonalności. Z tego względu nacisk na solidną architekturę stał się koniecznością.

Qubity

Podstawowym problemem przy tworzeniu emulatora jest symulacja zachowania qubitów. Pozornie trywialne zadanie szybko okazuje się być problematyczne. O ile pojedynczy qubit daje się zasymulować dość łatwo, o tyle dwa lub więcej kwantowych bitów wchodzących ze sobą w interakcje rodzi zupełnie nowe problemy. Obiektowe podejście do projektowania pozwoliło jednak stawić im czoła.

Model qubitu zaimplementowany w Quantum Shell składa się z wektora stanów (będącego macierzą liczb zespolonych) oraz listy powiązanych qubitów. Aby uniknąć przekłamań, qubity wchodzące ze sobą w interakcje tworzą jeden wspólny stan. Wszystkie qubity z jednego, spójnego stanu posiadają referencję na wspólny wektor stanu. Takie rozwiązanie zapewnia, że operacje wykonywane na tych qubitach będą zgodne z teorią obliczeń kwantowych (np. nigdy nie wykonamy operacji na pojedynczym qubicie ze splątanej grupy).

Przykładowa tablica (*StateVector*) stanów dla pojedynczego niepowiązanego qubita może wyglądać następująco:

$$[1 \;\; 0]$$

Powyższy stan ukaże się po wyświetleniu wektora stanów dla nowo zainicjalizowanego qubita (ponieważ domyślnym stanem jest |0⟩). Indeks w wektorze oznacza stan, a wartość umieszczona pod tym indeksem jest amplitudą stanu.

Metoda *JoinState* umożliwia łączenie stanów kilku qubitów. Przykładowo dla trzech qubitów ustawionych na |0⟩, jeżeli połączymy ich stan w jeden, wspólny, to otrzymamy wektor stanu:

[1 0 0 0 0 0 0 0]

Czyli odczytanie stanu |000⟩ jest zdarzeniem pewnym.

Oprócz referencji na wektor stanu każdy qubit przechowuje referencję na listę qubitów, z którymi jest powiązany. Dla danego qubitu *Q* jego lista powiązanych *StateQubitList* zawsze zawiera przynajmniej jeden wpis – *Q*. Ponieważ każdy qubit zawiera swój unikalny indeks (*QubitIndex*) lista *StateQubitList* jest zawsze posortowana. Dzięki zachowaniu kolejności qubitów zawsze można rozróżnić poszczególne qubity we wspólnym stanie.

Bramki kwantowe i przekształcenia stanu

Chcąc zachować jak największą uniwersalność utworzonego modelu zdecydowałem się na macierzową reprezentację bramek kwantowych. Zostały one zaimplementowane jako klasy dziedziczące po abstrakcyjnej klasie bazowej *QuantumGate*. Każdy operator zawiera dwuwymiarową macierz zespoloną reprezentującą wykonywaną przez niego transformację. Jeżeli było to możliwe, bramki były definiowane w swoich konstruktorach. Pozostałe (kwantowa transformata Fouriera i jej odwrotna wersja) były tworzone w konstruktorze dla wymaganego wymiaru.

Operacje wykonywane na stanie kwantowym są realizowane macierzowo. Qubit posiada metodę umożliwiającą przekształcenie stanu (*TransformState*), która pobiera obiekt bramki kwantowej. Następnie przelicza własny stan jako iloczyn macierzowy wektora stanu z macierzą transformacji wykonywanej przez bramkę. Dodatkowo, jeżeli qubit przechowuje wspólny stan grupy, operator macierzowy zostanie odpowiednio rozszerzony (poprzez tensoryzację) tak, aby wykonać operację identyczności na qubitach nie będących bezpośrednim celem transformacji. Tym samym zawsze przekształcany jest cały stan – uniwersalna metoda zachowa się odpowiednio zarówno dla pojedynczego qubitu jaki i dla splątanej grupy.

Przykładowa aplikacja bramki Hadamarda do pierwszego z dwóch qubitów o wspólnym stanie mogłaby być zapisana następująco:

Qubit Q1 znajduje się we wspólnym stanie z Q2:

[1 0 0 0]

Chcemy do niego zaaplikować bramkę Hadamarda, której macierzowa reprezentacja jest postaci:

$$\begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{bmatrix}$$

Jednak bramka ta operuje wyłącznie na jednym qubicie. Nie możemy więc obliczyć przekształconego stanu jako zwyczajny iloczyn wektora stanu z powyższą macierzą. Możemy ją jednak odpowiednio rozszerzyć tak, aby dla Q2 została zaaplikowana macierz identyczności:

$$I \otimes H = \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 0 & 0 \\ \frac{1}{\sqrt{2}} & \frac{-1}{\sqrt{2}} & 0 & 0 \\ 0 & 0 & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ 0 & 0 & \frac{1}{\sqrt{2}} & \frac{-1}{\sqrt{2}} \end{bmatrix}$$

Teraz możemy użyć powyższego operatora aby obliczyć końcowy stan:

$$[1\ 0\ 0\ 0] \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 0 & 0 \\ \frac{1}{\sqrt{2}} & \frac{-1}{\sqrt{2}} & 0 & 0 \\ 0 & 0 & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ 0 & 0 & \frac{1}{\sqrt{2}} & \frac{-1}{\sqrt{2}} \end{bmatrix} = [\frac{1}{\sqrt{2}}\ \frac{1}{\sqrt{2}}\ \ 0\ \ 0]$$

Ponieważ dwa pierwsze miejsca w wektorze stanu odpowiadają stanom |00⟩ i |01⟩ (czyli 0 i 1 binarnie) widzimy, że nastąpiło przekształcenie:

$$|00\rangle \rightarrow |00\rangle + |01\rangle$$

Tym samym pierwszy qubit został ustawiony w superpozycję stanów 0 i 1 o równych amplitudach. Zastosowanie wektorowej reprezentacji wspólnego stanu kwantowego oraz macierzowej reprezentacji bramek kwantowych wraz z opcją tensoryzacji tworzy bardzo elastyczny model symulacji obliczeń kwantowych.

Podglądanie stanu

Z uwagi na cel aplikacji, którym jest lepsze zrozumienie idei obliczeń kwantowych, dodano możliwość podejrzenia aktualnego stanu bez dokonywania pomiaru. Qubit posiada metodę *Peek*, która zwraca łańcuch znaków reprezentujący stan (wektory w zapisie bra-ket poprzedzane ich zespolonymi amplitudami). Jeżeli qubit posiada referencję na wspólny stan grupy, podglądnięcie jego stanu zwróci reprezentację stanu

całej grupy. Takie rozwiązanie zapewnia, że odczytywany stan jest zawsze pełny (m.in. nie dojdzie do wyświetlenia wartości pojedynczego qubitu ze splątanej pary).

Pomiar stanu

Chcąc oddać probabilistyczny charakter obliczeń kwantowych, podczas pomiaru wykorzystywane są pseudolosowe współczynniki. Obliczane jest prawdopodobieństwo odczytania wartości zero dla badanego qubitu (jeżeli jest on częścią grupy to zostaną uwzględnione wszystkie możliwe stany), a następnie pseudolosowy współczynnik z przedziału $(0, 1)$ wpływa na decyzję o faktycznie zwróconej wartości (jeżeli jest on mniejszy lub równy obliczonej wartości prawdopodobieństwa – zostanie zwrócone zero). Po odczycie cały stan (niezależnie czy qubit występuje w pojedynkę czy w grupie) jest aktualizowany i normalizowany.

Aktualizacja zostaje dokonana poprzez zerowanie stanów niemożliwych do odczytania po dokonaniu pomiaru (a tym samym – ustaleniu) wartości jednego z qubitów. Mając stan z poprzedniego przykładu:

$$[\frac{1}{\sqrt{2}}\ \frac{1}{\sqrt{2}}\ \ 0\ \ 0]$$

Odczytanie drugiego z qubitów nie będzie miało wpływu na cały stan – odczytanie 0 jest zdarzeniem pewnym, ponieważ amplitudy stanów $|10\rangle$ i $|11\rangle$ wynoszą 0. Jednak odczytanie pierwszego z qubitów na pewno zmieni wektor całego stanu. Jeżeli pomiar zwróci jeden, amplituda stanu $|00\rangle$ będzie musiała zostać wyzerowana. Aktualizacja przemieniłaby zatem powyższy stan w następujący:

$$[0\ \frac{1}{\sqrt{2}}\ \ 0\ \ 0]$$

Oczywiście jest to stan niestabilny, gdyż suma prawdopodobieństw odczytania stanów nie wynosi 1. Z tego powodu po aktualizacji następuje normalizacja stanu. Obliczany jest współczynnik normalizacji będący pierwiastkiem sumy kwadratów wszystkich amplitud stanu, a następnie każda amplituda jest przez niego dzielona. W naszym przykładzie współczynnik normalizacji byłby następujący:

$$normalizer = \sqrt{0^2 + \left(\frac{1}{\sqrt{2}}\right)^2 + 0^2 + 0^2} = \sqrt{\frac{1}{2}} = \frac{1}{\sqrt{2}}$$

Po normalizacji (podzieleniu wszystkich amplitud) stan całego układu zostałby pozostawiony w postaci:

$$[0\ \ 1\ \ 0\ \ 0]$$

Powyższy stan jest poprawny, gdyż suma prawdopodobieństw odczytania poszczególnych stanów wynosi 1. Jest to również wynik oczekiwany: skoro wiemy, że drugi z qubitów podczas pomiaru na pewno zwróci 0, a pierwszy z nich odczytaliśmy jako 1, jedynym możliwym do odczytania stanem jest $|01\rangle$.

Kontrolowane operacje kwantowe

Wybranie macierzowej metody przeliczania stanu kwantowego przynosi bardzo wiele korzyści, jednak stwarza także pewne problemy. Ponieważ typowe macierze kontrolowanych operatorów kwantowych wymagają, aby qubity biorące udział w przekształceniu były reprezentowane przez wspólny wektor, pojawił się problem z implementacją bramek operujących na qubitach nie będących swoimi sąsiadami. Zamiana qubitów miejscami wewnątrz stanu nie wchodziła w grę, ponieważ spowodowałaby ona przekłamania w wektorze stanu (poszczególne bity w każdym ze stanów zostałyby zamienione miejscami).

Z tych powodów powstała operacja budująca kontrolowany operator kwantowy dla dowolnej jedno-qubitowej bramki. Qubit posiada metodę *TransformStateControlled* pobierającą bramkę oraz qubit kontrolny. Następnie budowany jest operator mogący zostać użyty na całym stanie kwantowym. Wykonuje on operację identyczności na wszystkich qubitach, których przekształcenie nie dotyczy, a qubit docelowy zostaje przekształcony w zależności od qubitu kontrolnego.

Tym samym zamiast budować specjalizowane dwu-qubitowe bramki kontrolowane qubit używa jednej metody pozwalającej mu utworzyć kontrolowaną wersję dla każdej jedno-qubitowej bramki (np. bramka CNOT może zostać utworzona jako kontrolowana bramka X – *PauliXGate*).

Skierowane operacje kwantowe

W algorytmach kwantowych często wykorzystywany jest pewien specyficzny rodzaj przekształceń, gdzie jeden z qubitów nie tylko kontroluje, czy pewna operacja zostanie wykonana, ale bierze udział w przekształceniu drugiego z qubitów. Przykładem może być bramka znana z algorytmu Deutscha:

$$|x\rangle|y\rangle \rightarrow |x\rangle|y \oplus f(x)\rangle$$

Chociaż teoretycznie można dobrze przybliżyć działanie takiej bramki używając zestawu uniwersalnego, uznałem to za bardzo niewygodne i niepraktyczne. Dlatego została zaimplementowana operacja *TransformRegisterStateDirected*[1]. Umożliwia ona realizację bramek operujących na rejestrach kwantowych. Jako argument przyjmuje funkcję transformującą argument y (w powyższym przykładzie jest to operacja XOR zapisana $\oplus$) oraz funkcję odwzorowującą x (powyżej - $f(x)$), a także pierwszy qubit rejestru kontrolnego. Może być to pojedynczy qubit, a może być cały rejestr. Wymagane jest jednak, aby rozmiar rejestru kontrolnego był większy lub równy rozmiarowi rejestru docelowego (co jest zgodne z zastosowaniem bramek tej postaci)

Rejestry, na których operuje bramka skierowana muszą do siebie przylegać (tzn. indeks ostatniego qubitu rejestru docelowego musi być dokładnie o jeden mniejszy od indeksu pierwszego qubita rejestru kontrolnego). Jeżeli pewne wymagania co do stanów wyjściowych nie zostały spełnione operacja rzuci wyjątkiem z opisem problemu.

---

[1] Nazwa skierowana operacja (*DirectedTransform*) zotała wprowadzona przeze mnie, ponieważ moim zdaniem dobrze oddaje charakter bramki (pierwszy qubit zostaje *skierowany* do drugiego, umożliwiając pewne przekształcenia).

Należy również zwrócić uwagę, że możliwość dostarczania funkcji operujących na rejestrach niesie bardzo duże możliwości, ale też pewne zagrożenia. Ponieważ przekształcenia te zapisane są w kodzie, nie mamy gwarancji ich unitarności. Z tego powodu qubity zostały wyposażone w pewne zabezpieczenie – jeżeli stan qubitu zostanie zniszczony, qubit podczas odczytu może zwrócić wartość -1, która informuje o błędzie.

Implementacja liczb zespolonych

Z powodu chęci dopasowania implementacji liczb zespolonych i macierzy liczb zespolonych do własnych potrzeb zdecydowałem się utworzyć klasy *Complex* oraz *ComplexMatrix* udostępniające stosowne metody.

Pierwsza z nich przechowuje części rzeczywistą i urojoną jako zmiennoprzecinkowe liczby podwójnej precyzji. Zaimplementowane zostały również standardowe operacje arytmetyczne. Udostępnione są trzy konstruktory, jedno, dwu i trzy-argumentowy. Programista może utworzyć reprezentację liczby urojonej podając wyłącznie część rzeczywistą (urojona zostaje ustawiona na 0) lub zarówno rzeczywistą jak i urojoną. Trzeci parametr jest opcjonalny i informuje o precyzji (liczbie istotnych miejsc po przecinku). Został on wprowadzony aby zapobiec wpływowi niechcianych wartości na dalekich miejscach po przecinku na końcowe wyniki (który stawał się bardzo wyraźny przy korzystaniu z kwantowej transformaty Fouriera).

Druga z klas, *ComplexMatrix*, przechowuje listę list liczb zespolonych, oraz udostępnia metody *Dot* oraz *Tensorize*, pozwalające łatwo wykonywać stosowne przekształcenia.

2. Interpreter

W celu ułatwienia dostępu do emulatora wykonany został prosty interpreter, udostępniający własny zestaw instrukcji. Mogą one być zarówno prowadzane ręcznie, na bieżąco, jak również zapisane w postaci skryptu. Interpreter pozwala na załadowanie skryptu poprzez komendę *load*, po której wprowadzeniu użytkownik poproszony zostaje o wprowadzenie ścieżki do pliku tekstowego. Komenda *help* spowoduje wypisanie dostępnych operacji.

Podstawowe komendy obejmują między innymi pomiar (*Measure*), podglądnięcie stanu (*Peek*) czy przekształcenie przez określoną bramkę (*H(qubitNumber), X(qubitNumber)* itd.). Możliwe jest również wykonywanie operacji kontrolowanych (wpisanie *c-B*, gdzie *B* jest symbolem bramki, spowoduje przekształcenie qubita podanego jako drugi argument przez bramkę *B* kontrolowaną przez qubita podanego jako pierwszy argument).

Przykładowy skrypt może wyglądać następująco (uwaga: komentarze w skrypcie mają wyłącznie charakter opisowy na rzecz tego opracowania i nie mogą występować w skryptach wykonywanych na emulatorze Quantum Shell):

```
Peek(0)         //podejrzyj stan qubita o indeksie 0.
Set(0,1)        //ustaw qubit o indeksie 0 na stan 1.
Peek(0)         //podejrzyj stan qubita o indeksie 0.
Set(2,1)        //ustaw qubit o indeksie 2 na stan 1.
Peek(2)         //podejrzyj stan qubita o indeksie 2.
Join(0-7)       //utwórz wspólny stan dla qubitów o indeksach 0–7.
Peek(7)         //podejrzyj stan qubita o indeksie 7.
c-X(0,2)        //wykonaj CNOT na qubicie 2 kontrolowanym przez 0.
Peek(7)         //podejrzyj stan qubita o indeksie 7.
exit            //zakończ.
```

Dodatkowo zostało wprowadzone menu *Example*, umożliwiające demonstrację wykonania algorytmów kwantowych procedur (kod wszystkich przykładów dostępny jest w projekcie, w pakiecie *Examples*).

3. Algorytmy

Po uruchomieniu interpretera możemy wprowadzić komendę *Examples*, która spowoduje przejście do menu o tej nazwie. Do zasymulowania dostępne są cztery algorytmy:

- Kwantowa podprocedura algorytmu faktoryzacji Shora (dla problemu określania rzędu modulo),
- Kwantowa podprocedura algorytmu Simona (dla problemu określania ukrytego ciągu),
- Kwantowa procedura algorytmu Deutscha,
- Kwantowa procedura algorytmu Deutscha – Jozsy.

Aby uruchomić wybraną procedurę należy wpisać jej nazwę z menu *Examples*.

Algorytm faktoryzacji

Problem faktoryzacji został przedstawiony na przykładzie szukania rzędu 7 *mod* 25. Symulowana procedura kwantowa przebiega następująco:

```
Reset()                 //Wykorzystano rejestr 10-qubitowy.
Join(0-4)
X(0)
Join(5-9)
QFT(5)
TransformDirected(5, 4) //|x>|y> -> |x>|y*(7^x) mod 25>.
IQFT(5)
Measure(9)
Measure(8)
Measure(7)
Measure(6)
Measure(5)
```

Ponieważ interpreter operuje na ośmio-qubitowym rejestrze nie jest możliwe wykonanie powyższego przykładu (nie można zapisać przykładowej liczby 25 na mniej

niż pięcio-qubitowym rejestrze). Bramka TransformDirected nie jest dostępna bezpośrednio z poziomu interpretera (dostępny jest jej kod w *Examples.Factorization*).

Pomiar rejestru zwraca liczbę x taką, że $\frac{x}{2^n}$ (tu: $n = 5$ to rozmiar rejestru kontrolnego) jest równe $\frac{k}{r}$, gdzie r jest poszukiwanym rzędem 7 mod 25, a k pewną losową liczbą całkowitą z przedziału $[0, r-1]$. Procedurę należy powtórzyć aż odczytamy wynik inny niż 0. W naszym wypadku odczytane zostanie $|10000\rangle$ czyli 16. Możemy więc zinterpretować wynik:

$$\frac{16}{2^5} = \frac{1}{2}$$

Mamy tu do czynienia z przypadkiem gdzie k i r miały pewien nietrywialny wspólny dzielnik, przez który zostały skrócone. Wartość k wynosi 4 ($7^4 mod\ 25 = 1$), można ją poznać korzystając m.in. z algorytmu ułamków łańcuchowych. Niemniej widzimy, że algorytm działa poprawnie.

Istotnym jest fakt, że rozmiar naszego rejestru ogranicza możliwości wyboru liczb do faktoryzacji nie tylko ze względu na możliwość zapisu binarnego. Rozmiar rejestru kontrolnego (równy rozmiarowi rejestru docelowego) n musi spełniać $2^n \geq 2r^2$, więc dla $n = 5$ nasze r jest ograniczone przez 4 (ponieważ $2^5 = 32 = 2r^2$ dla $r = 4$).

Algorytm Simona

Problem ukrytego ciągu rozwiązywany przez algorytm Simona został zaprezentowany na przykładzie pewnej funkcji f spełniającej $f(x) = f(y) \Leftrightarrow x = y\ lub\ x = y \oplus s$ dla $s = 1010b$. Symulowana procedura kwantowa przebiega następująco:

```
Reset()                  //Wykorzystano rejestr 8-qubitowy.
Join(0-3)
Join(4-7)
H(4)
H(5)
H(6)
H(7)
TransformDirected(4, 3) //|x>|y> -> |x>|y XOR f(x)>, hardcoded f.
H(4)
H(5)
H(6)
H(7)
Measure(7)
Measure(6)
Measure(5)
Measure(4)
```

Z powodu swojej specyfiki bramka TransformDirected nie jest dostępna bezpośrednio z poziomu interpretera (dostępny jest jej kod w *Examples.HiddenSubgroup*).

Pomiar rejestru zwraca pewien ciąg binarny z taki, że $zs = 0$. W celu odnalezienia s należy odczytać $n-1$ (tu: $n = 4$) różnych ciągów z i rozwiązać układ równań postaci

$sz = 0\ (mod\ 2)$. W przykładzie, uruchamiając rejestr kilka razy możemy odczytać wartości m.in. $|0001\rangle, |0101\rangle, |1011\rangle$. Układ równań ułożony z wykorzystaniem tych wartości ma dwa możliwe rozwiązania: $s = 0000$ oraz $s = 1010$. Zgodnie z algorytmem wybieramy rozwiązanie niezerowe. Tym samym odczytaliśmy ukryty ciąg *s*.

Algorytm Deutscha – Jozsy i Algorytm Deutscha

Problem Deutscha – Jozsy rozwiązywany jest na przykładzie dwóch funkcji: stałej $f(x) = 0$ oraz zbalansowanej $f(x) = x\ mod\ 2$. Symulowana procedura kwantowa dla jednej z funkcji $f(x)$ przedstawia się następująco:

```
Reset()                  //Wykorzystano rejestr 8-qubitowy.
X(0)
H(0)
Join(1-7)
H(1)
H(2)
H(3)
H(4)
H(5)
H(6)
H(7)
TransformDirected(1, 0) //|x>|y> -> |x>|y XOR f(x)>.
H(1)
H(2)
H(3)
H(4)
H(5)
H(6)
H(7)
Measure(7)
Measure(6)
Measure(5)
Measure(4)
Measure(3)
Measure(2)
Measure(1)
```

Powyższa procedura dla algorytmu Deutscha różni się jedynie ograniczeniem qubitów do tych o indeksach 0 oraz 1 (działamy na kontrolnym qubicie, zamiast całego rejestru).

Z powodu swojej specyfiki bramka TransformDirected nie jest dostępna bezpośrednio z poziomu interpretera (dostępny jest jej kod w *Examples.DeutschJozsa*, a kod prostszej wersji dla algorytmu Deutscha w *Examples.Deutsch*).

Dla funkcji stałej pomiar rejestru zawsze zwraca wyłącznie zera ($|0000000\rangle$ dla algorytmu Deutscha – Jozsy lub $|0\rangle$ dla algorytmu Deutscha). Dla funkcji zbalansowanej w odczytanym wyniku pojawi się przynajmniej jedna jedynka. Algorytmy zwracają poprawne wyniki i potrafią zidentyfikować charakter funkcji (czy jest stała czy zbalansowana).