Matematyka w Informatyce Przyszłości

Sprawozdanie z laboratoriów

Krzysztof Romanowski

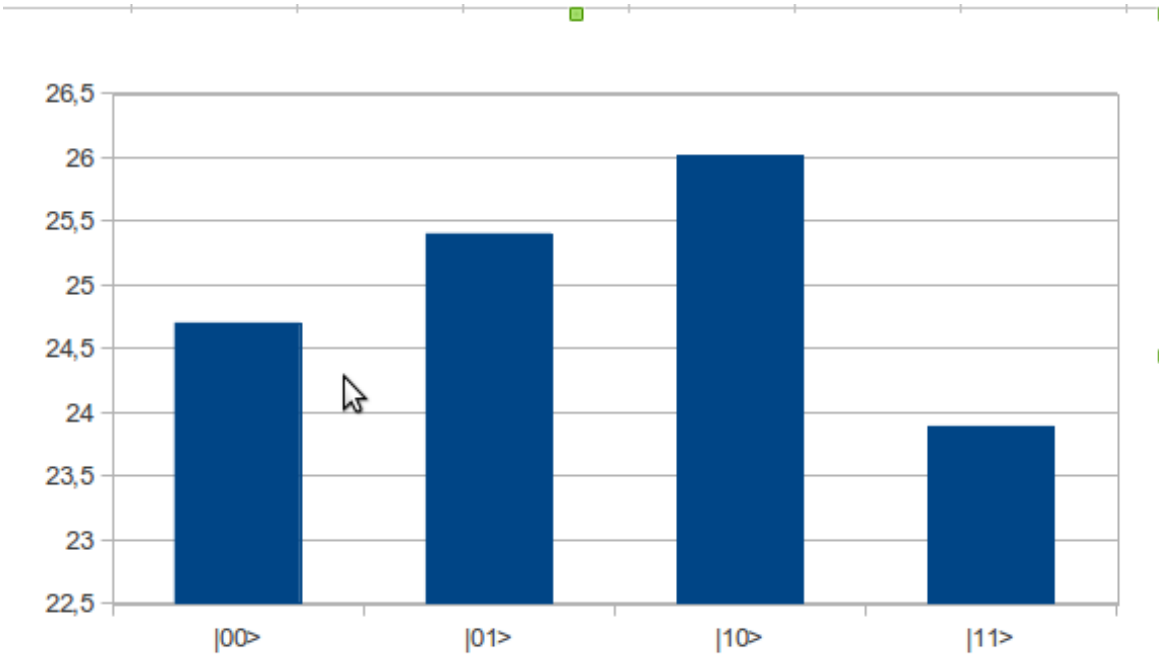
Labolatorium 1

1. Podstawowe pomiary

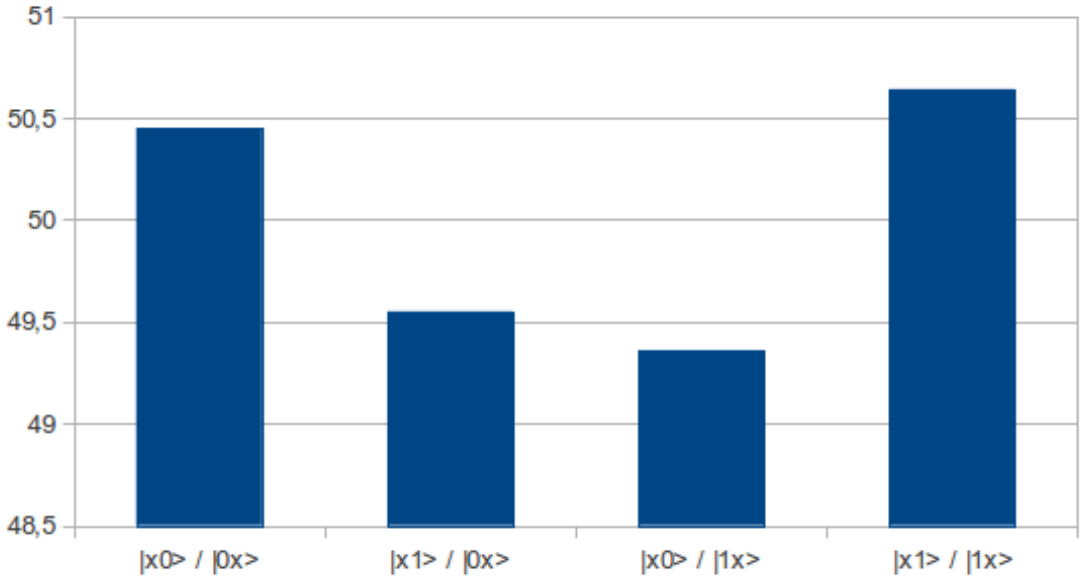
```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <quantum.h>

int main ()
{
  quantum_reg reg;
  int result, result1, result2;
  int results[4];
  int i = 0;
  for(i=0;i<4;i++){
     results[i]=0;
  }

  srand(time(0));



  for(i=0;i<20000;i++){
       reg = quantum_new_qureg(0, 2);

       quantum_walsh(2, &reg);


       result1 = 0;
       result2 = 0;
       result = quantum_measure(reg);
       results[result] += 1;

       printf("%i\n", result);
  }

  printf("%i %i %i %i\n", results[0], results[1], results[2],
results[3]);

  return 0;
}
```

Po dokonaniu reprezentatywnej ilości pomiarów (20000) otrzymaliśmy następujące wynki:

Rozkład otrzymanych wyników:



oraz prawdopodobieństwa warunkowe:



Wnioski:
Wyniki otrzymane na poszczególnych Qubitach są od siebie niezależne.

Bramka swap

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <quantum.h>

int main ()
{
  quantum_reg reg;
  int result;
  int i = 0;

  srand(time(0));


  for(i=0;i<4;i++){
        reg = quantum_new_qureg(i, 2);

        quantum_cnot(0,1, &reg);
        quantum_cnot(1,0, &reg);
        quantum_cnot(0,1, &reg);

        result = quantum_measure(reg);

        printf("%i -> %i\n", i, result);
  }

  return 0;
}
```

Bramka skonstruowana z bramek CNOT daje dokładne i poprawne rezultaty.

00 -> 00
01 -> 10
10 -> 01
11 -> 11

Bramka Toffoliego

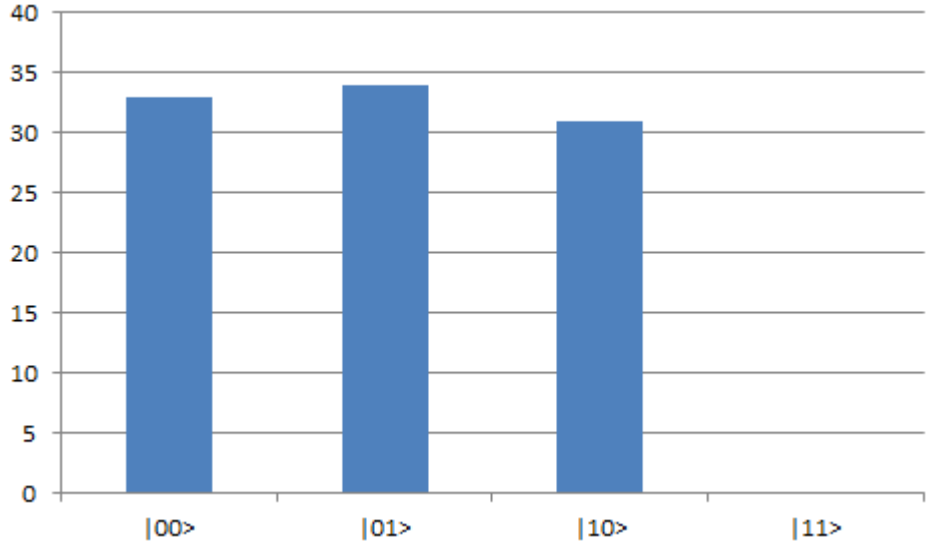
```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <quantum.h>

int main ()
{
  quantum_reg reg;
  int result;
  int i = 0;

  srand(time(0));


  for(i=0;i<8;i++){
        reg = quantum_new_qureg(i, 3);
        quantum_toffoli(0, 1, 2, &reg);
        result = quantum_measure(reg);
        printf("%i -> %i\n", i, result);
  }

  return 0;
}
```

Testowaliśmy wyniki funkcji quantum_toffoli z libquantum. Wyniki (zgodne z oczekiwaniami):
000 -> 000
001 -> 001
010 -> 010
011 -> 011
100 -> 100
101 -> 101
110 -> 111
111 -> 110

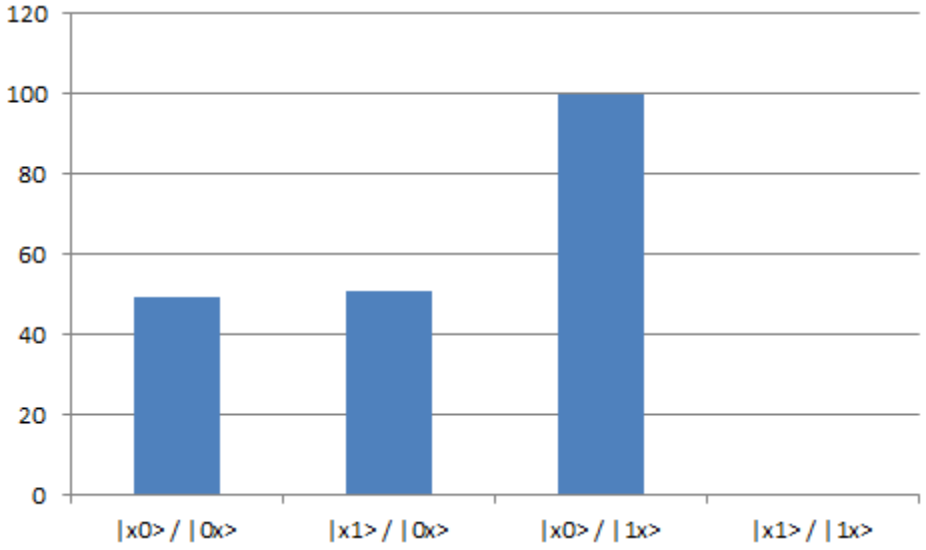
Lab2

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <quantum.h>

int main ()
{
  quantum_reg reg;

  srand(time(0));

  int c1 = 0;
  int c2 = 0;
  int c3 = 0;
  int c4 = 0;
  int i;
  for(i = 0; i<1000; i++){
  reg = quantum_new_qureg(0, 2);
  quantum_r_y(1,2*0.6154,&reg);
  quantum_hadamard(0,&reg);
  quantum_r_y(0,3.14/4.0,&reg);
  quantum_cnot(1,0,&reg);
  quantum_r_y(0,-3.14/4.0,&reg);


  int result = quantum_measure(reg);
  if(result == 0)
   c1++;
  if(result == 1)
   c2++;
  if(result == 2)
    c3++;
  if(result == 3)
    c4++;
  }
  printf("0) %d\n",c1);
  printf("1) %d\n",c2);
  printf("2) %d\n",c3);
  printf("3) %d\n",c4);
  return 0;
```

Prawdopodobieństwo wystąpienia poszczególnych stanów:



Prawdopodobieństwo warunkowe wystąpienia poszczególnych stanów:



Wnioski:
W tym przypadku okazało się, że stany na poszczególnych qbitach są od siebie zależne. Jeżeli na pierwszym qbicie występuje 0 to, rozkład stanów drugiego qbitu jest równomierny. Jeżeli natomiast na pierwszym qbicie występuje 1, to na drugim zawsze będzie 0.

Problem Deutscha

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <quantum.h>

void uf0(quantum_reg * reg);
void uf1(quantum_reg * reg);
void uf2(quantum_reg * reg);
void uf3(quantum_reg * reg);

void deutch_problem(quantum_reg * reg);

int main ()
{
  quantum_reg reg;
  int i;
  int result1;
  for (i = 0; i < 4; i++)
  {
      reg = quantum_new_qureg(i, 2);
      deutch_problem(&reg);
      result1 = quantum_measure(reg);
      printf("deutch(%d) = %d\n", i, result1);
  }

  return 0;
}

void uf0(quantum_reg * reg)
{}

void uf1(quantum_reg * reg)
{
     quantum_cnot(0, 1, reg);
}

void uf2(quantum_reg * reg)
{
     quantum_sigma_x(1, reg);
     quantum_cnot(0, 1, reg);
}

void uf3(quantum_reg * reg)
{
     quantum_sigma_x(1, reg);
}

void deutch_problem(quantum_reg * reg)
{
```

```
    quantum_sigma_x(0, reg);
    quantum_sigma_x(1, reg);
  //quantum_cnot(0, 1, &reg);
  //quantum_cnot(1, 0, &reg);
  //quantum_cnot(0, 1, &reg);


    quantum_hadamard(0, reg);
    quantum_hadamard(1, reg);

    uf2(reg);

    quantum_hadamard(0, reg);


}
```

Problem rozwiązaliśmy korzystając z dwubitowego rejestru. Na młodszym bicie trzymamy wejście na starszym wyjście.

funkcja	|x0>	|x1>
f_0	|0x>	|0x>
f_1	|0x>	|1x>
f_2	|1x>	|0x>
f_{3x}	|1x>	|1x>

Widzimy, że berstein verizani problemfunkcje działają poprawnie.
Przejdzmy do własciwego zadania.

Przekształcenie	Wyjście
U_{f0}	|1>
U_{f1}	|0>
U_{f2}	|0>
U_{f3}	|1>

Widzimy że poprawnie rozwiązaliśmy zadany problem.

Problem Bernsteina-Vaziraniego

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <quantum.h>

void quantum_uf (int a, quantum_reg *reg) {

	int i;
	for (i = reg->width - 1; i; --i) if (a & 1 << (i - 1)){
		if (a & 1 << (i - 1))
			quantum_cnot(i, 0, reg);
	}

}


void printa(int a, int n) {

	int i;
	printf("a = ");
	for (i = n - 1; i >= 0; --i)
		printf(a & 1 << i ? "1" : "0");

	printf("\n");
}

void quantum_uf_test(int a, int x, int n) {

	quantum_reg reg = quantum_new_qureg(x << 1, n + 1);
	printa(a, n);
	printf("Uf");

	quantum_print_qureg(reg);
	quantum_uf(a, &reg);
	printf("---->");

	quantum_print_qureg(reg);
	quantum_delete_qureg(&reg);
}

int main () {
```

```
	int a, n, m; quantum_reg reg;
	srand(time(0));
	printf("=============== DZIAŁANIE FUNKCJI Uf ================\n\n");
	quantum_uf_test(116, 113, 7);
	quantum_uf_test(116, 115, 7);
	quantum_uf_test(115, 113, 7);
	printf("========== ROZWIĄZANIE KLASYCZNE (n wywołań Uf) ==========\n\n");
	a = 500;
	n = 10;

	printa(a, n);
	for (m = 0; m < n; ++m) {
		reg = quantum_new_qureg(1 << (m + 1), n + 1);
		quantum_uf(a, &reg);
		printf("a%i:", m);
		quantum_print_qureg(reg);
		quantum_delete_qureg(&reg);
	}
	printf("========= ROZWIĄZANIE KWANTOWE (1 wywołanie Uf) ==========\n\n");
	printa(a, n);

	reg = quantum_new_qureg(1, n + 1);

	for (m = 0; m < n + 1; ++m){
		quantum_hadamard(m, &reg);
	}

	quantum_uf(a, &reg);

	for (m = 0; m < n + 1; ++m){
		quantum_hadamard(m, &reg);
	}

	printf("a:");

	quantum_print_qureg(reg);
	quantum_delete_qureg(&reg);
	return 0;
}
```

3 testy funkcji Uf - parametr a - 116, oraz 115, inicjalizacja rejestru - parametr x - 226, oraz 230

```
========= ROZWIĄZANIE KWANTOWE (1 wywołanie Uf) ==========

a = 0111110100
a: 1.000000 +0.000000i|1001> (9.999998e-01) (|011 1110 1001>)
```