In [1]:
```
import numpy as np
from matplotlib import pyplot as plt
from numpy import linalg as LA
from scipy import linalg as sLA
import time

d = 2
```

In [2]:
```
a = np.array([1,0])
b = np.array([[1,0]]).T
print(b.shape)

np.kron(b,a)
```

Out[2]:
```
(2, 1)
array([[1, 0],
       [0, 0]])
```

In [3]:
```
class Qstate():
    def __init__(self, N = 2):
        self.A_dims = []
        self.B_dims = []
        self.tensor = np.zeros(tuple([2]*N))
        self.vec = np.zeros(2**N)
        self.N = N
        self.rho = None


    def build_pure_random_state(self): # строит случайное состояние заданно
#         vec = np.random.randn(*([2]*self.N)) + 1j*np.random.randn(*([2]*
#         self.tensor = vec

        vec = np.random.randn(2**self.N) + 1j*np.random.randn(2**self.N)
        self.vec = vec/np.sqrt(np.sum(vec*vec.conj()))
#         self.tensor = np.reshape(self.vec, [2]*self.N)
        self.rho = np.kron(np.reshape(self.vec, (self.vec.shape[0], 1)), s

    def build_state(self, vec):
        self.vec = vec
        #         self.tensor = np.reshape(self.vec, [2]*self.N)
        self.rho = np.kron(np.reshape(self.vec, (self.vec.shape[0], 1)), s


    def apply_U(self, U):
        self.vec = np.dot(U,self.vec)
#         self.tensor = np.reshape(self.vec, [2]*self.N)
        self.rho = np.kron(np.reshape(self.vec, (self.vec.shape[0], 1)), s

    def get_tensor(self): # возвращаем вектор
        return self.tensor

    def get_vec(self): # возвращаем вектор
        return self.vec

    def get_rho(self): # возвращаем матрицу плотности
        return self.rho
```

In [4]:
```
# Функция базового преобразования

def R_matrix(delta =0, theta = 0, phi = 0):
    R = np.zeros((2,2)).astype('complex')
    R[0][0] = np.cos(delta/2) - 1j*np.cos(theta)*np.sin(delta/2)
    R[1][0] = -1j*np.sin(theta)*np.sin(delta/2)*np.exp(1j*phi)
    R[0][1] = -1j*np.sin(theta)*np.sin(delta/2)*np.exp(-1j*phi)
    R[1][1] = np.cos(delta/2) + 1j*np.cos(theta)*np.sin(delta/2)
    return R
```

In [5]:
```
np.trace(R_matrix(-np.pi/4, np.pi/2, np.pi/2))
```

Out[5]:
```
(1.8477590650225735+0j)
```

In [6]:
```
# Проверим работу
eps = 0
R = np.kron(R_matrix(-np.pi/4, np.pi/2, np.pi/2), R_matrix(np.pi/20, np.pi,
R
```

Out[6]:
```
array([[ 9.21031520e-01+1.89218337e-17j,  1.83850077e-18-7.24867527e-02j,
         3.81503747e-01+2.15218664e-17j,  1.83850077e-18-3.00249961e-02j],
       [ 1.83850077e-18-7.24867527e-02j,  9.21031520e-01+2.77989006e-17j,
         1.83850077e-18-3.00249961e-02j,  3.81503747e-01+2.51988679e-17j],
       [-3.81503747e-01+2.51988679e-17j,  1.83850077e-18+3.00249961e-02j,
         9.21031520e-01-2.77989006e-17j, -1.83850077e-18-7.24867527e-02j],
       [ 1.83850077e-18+3.00249961e-02j, -3.81503747e-01+2.15218664e-17j,
        -1.83850077e-18-7.24867527e-02j,  9.21031520e-01-1.89218337e-17j]])
```

In [7]:
```
# Зададим случайное чистое состояние

state = Qstate(d)
state.build_pure_random_state()
state.get_vec()
```

Out[7]:
```
array([ 0.16051033+0.42447866j, -0.0799487 -0.43693425j,
       -0.1208129 +0.70344918j, -0.06609776+0.28800268j])
```

In [20]:
```
# Функция из пункта 2

def get_prob(state, L = 1000, d = 2, noisy = True, loc = 0, scale = 0.1):

    result = 0

    zero_vec = np.zeros(2**d)
    zero_vec[0] = 1


    v = state.get_vec()

    for i in range(L):
        if noisy == True:
            eps = np.random.normal(loc=loc, scale=scale)
        else:
            eps = 0
        R = np.kron(R_matrix(-np.pi/4, np.pi/2, np.pi/2), R_matrix(np.pi/2(

        state.apply_U(R)
        vec = state.get_vec()
        state.build_state(v)
#         print(eps, state.get_vec())
        p = np.abs(np.dot(zero_vec, vec))**2
        if np.random.rand() < p:
            result += 1

    result = result/L
    return result

state = Qstate(d)
state.build_pure_random_state()

get_prob(state, noisy = False)
```

Out[20]: 0.183

In [ ]:
```

```

Вычислим хи матрицу процесса:

In [10]:
```
# Вычисление Xi-матрицы операции

def XI_matrix(U):
    XI0 = np.reshape(U, (U.shape[0]**2,1))
    XI1 = np.reshape(U, (1,U.shape[0]**2))

    XI = np.reshape(np.tensordot(XI0, XI1.conj(), [1,0]), (XI0.shape[0],XI(
    return XI
```

In [11]:
```
# Функция для вычисления усреднённой по ансамблю XI-матрицы (просто для эк

def weighted_sum(L = 100):
    XI_list = []
    p_list = []

    zero_vec = np.zeros(2**d)
    zero_vec[0] = 1

    for i in range(L):

        eps = np.random.normal(loc=0, scale=0.1)
        p = 1/(0.1*np.sqrt(2*np.pi))*np.exp(-eps**2/(2*0.1**2))
        p_list.append(p)
        R = np.kron(R_matrix(-np.pi/4, np.pi/2, np.pi/2), R_matrix(np.pi/2(
        XI = XI_matrix(R)
        XI_list.append(XI*p)

    p_list = np.array(p_list)
    XI_list = np.array(XI_list)

    XI_mean = np.mean(XI_list, axis = 0)

    return XI_mean
```

In [12]:
```
weighted_sum(L = 1000)
```

Out[12]:
```
array([[ 2.39656391e+00-1.09503379e-20j,  2.83261690e-05+1.88138907e-01j,
         9.92689275e-01-3.57342791e-17j,  1.17330834e-05+7.79296870e-02j,
         2.83261690e-05+1.88138907e-01j,  2.39641683e+00-7.17982174e-04j,
         1.17330834e-05+7.79296870e-02j,  9.92628352e-01-2.97397954e-04j,
        -9.92689275e-01-8.59854121e-17j, -1.17330834e-05-7.79296870e-02j,
         2.39656391e+00+1.21273738e-16j,  2.83261690e-05+1.88138907e-01j,
        -1.17330834e-05-7.79296870e-02j, -9.92628352e-01+2.97397954e-04j,
         2.83261690e-05+1.88138907e-01j,  2.39641683e+00-7.17982174e-04j],
       [ 2.83261690e-05-1.88138907e-01j,  1.47702219e-02-1.00324551e-36j,
         1.17330834e-05-7.79296870e-02j,  6.11802622e-03-2.19447937e-19j,
         1.47702219e-02-1.00324551e-36j, -2.83261690e-05-1.88138907e-01j,
         6.11802622e-03-2.19447937e-19j, -1.17330834e-05-7.79296870e-02j,
        -1.17330834e-05+7.79296870e-02j, -6.11802622e-03-5.29794186e-19j,
         2.83261690e-05-1.88138907e-01j,  1.47702219e-02+7.49242123e-19j,
        -6.11802622e-03-5.29794186e-19j,  1.17330834e-05+7.79296870e-02j,
         1.47702219e-02+7.49242123e-19j, -2.83261690e-05-1.88138907e-01j],
       [ 9.92689275e-01+3.57357348e-17j,  1.17330834e-05+7.79296870e-02j,
         4.11185361e-01+8.56079198e-22j,  4.86000226e-06+3.22795333e-02j,
         1.17330834e-05+7.79296870e-02j,  9.92628352e-01-2.97397954e-04j,
         4.86000226e-06+3.22795333e-02j,  4.11160126e-01-1.23186266e-04j,
        -4.11185361e-01-5.04132665e-17j, -4.86000226e-06-3.22795333e-02j,
         9.92689275e-01+8.59722613e-17j,  1.17330834e-05+7.79296870e-02j,
        -4.86000226e-06-3.22795333e-02j, -4.11160126e-01+1.23186266e-04j,
         1.17330834e-05+7.79296870e-02j,  9.92628352e-01-2.97397954e-04j],
       [ 1.17330834e-05-7.79296870e-02j,  6.11802622e-03+2.19447937e-19j,
         4.86000226e-06-3.22795333e-02j,  2.53416943e-03-1.06450980e-37j,
         6.11802622e-03+2.19447937e-19j, -1.17330834e-05-7.79296870e-02j,
         2.53416943e-03-1.06450980e-37j, -4.86000226e-06-3.22795333e-02j,
        -4.86000226e-06+3.22795333e-02j, -2.53416943e-03-3.10346249e-19j,
         1.17330834e-05-7.79296870e-02j,  6.11802622e-03+5.29794186e-19j,
        -2.53416943e-03-3.10346249e-19j,  4.86000226e-06+3.22795333e-02j,
         6.11802622e-03+5.29794186e-19j, -1.17330834e-05-7.79296870e-02j],
       [ 2.83261690e-05-1.88138907e-01j,  1.47702219e-02-1.00324551e-36j,
```

```
[-1.17330834e-05+7.79296870e-02j, -6.11802622e-03+5.29794186e-19j,
 -4.86000226e-06+3.22795333e-02j, -2.53416943e-03+3.10346249e-19j,
 -6.11802622e-03+5.29794186e-19j,  1.17330834e-05+7.79296870e-02j,
 -2.53416943e-03+3.10346249e-19j,  4.86000226e-06+3.22795333e-02j,
  4.86000226e-06-3.22795333e-02j,  2.53416943e-03+1.06450980e-37j,
 -1.17330834e-05+7.79296870e-02j, -6.11802622e-03+2.19447937e-19j,
  2.53416943e-03+1.06450980e-37j, -4.86000226e-06-3.22795333e-02j,
 -6.11802622e-03+2.19447937e-19j,  1.17330834e-05+7.79296870e-02j],
[-9.92628352e-01-2.97397954e-04j,  1.17330834e-05-7.79296870e-02j,
 -4.11160126e-01-1.23186266e-04j,  4.86000226e-06-3.22795333e-02j,
  1.17330834e-05-7.79296870e-02j, -9.92689275e-01+8.59722613e-17j,
  4.86000226e-06-3.22795333e-02j, -4.11185361e-01+5.04132665e-17j,
  4.11160126e-01+1.23186266e-04j, -4.86000226e-06+3.22795333e-02j,
 -9.92628352e-01-2.97397954e-04j,  1.17330834e-05-7.79296870e-02j,
 -4.86000226e-06+3.22795333e-02j,  4.11185361e-01-8.56079198e-22j,
  1.17330834e-05-7.79296870e-02j, -9.92689275e-01+3.57357348e-17j],
[ 2.83261690e-05-1.88138907e-01j,  1.47702219e-02-7.49242123e-19j,
  1.17330834e-05-7.79296870e-02j,  6.11802622e-03-5.29794186e-19j,
  1.47702219e-02-7.49242123e-19j, -2.83261690e-05-1.88138907e-01j,
  6.11802622e-03-5.29794186e-19j, -1.17330834e-05-7.79296870e-02j,
 -1.17330834e-05+7.79296870e-02j, -6.11802622e-03-2.19447937e-19j,
  2.83261690e-05-1.88138907e-01j,  1.47702219e-02+1.00324551e-36j,
 -6.11802622e-03-2.19447937e-19j,  1.17330834e-05+7.79296870e-02j,
  1.47702219e-02+1.00324551e-36j, -2.83261690e-05-1.88138907e-01j],
[ 2.39641683e+00+7.17982174e-04j, -2.83261690e-05+1.88138907e-01j,
  9.92628352e-01+2.97397954e-04j, -1.17330834e-05+7.79296870e-02j,
 -2.83261690e-05+1.88138907e-01j,  2.39656391e+00-1.21273738e-16j,
 -1.17330834e-05+7.79296870e-02j,  9.92689275e-01-8.59854121e-17j,
 -9.92628352e-01-2.97397954e-04j,  1.17330834e-05-7.79296870e-02j,
  2.39641683e+00+7.17982174e-04j, -2.83261690e-05+1.88138907e-01j,
  1.17330834e-05-7.79296870e-02j, -9.92689275e-01-3.57342791e-17j,
```

In [46]:

```
# Вычисление усреднённой по шуму с заданным распределением Xi-матрицы

def Integrate(e_min = -0.5, e_max = 0.5, N=100, loc = 0, scale = 0.1):
    e_array = np.linspace(e_min, e_max, N)
    de = e_array[1] - e_array[0]
    I = 0

    for e in e_array:
        p  = 1/(scale*np.sqrt(2*np.pi))*np.exp(-(loc - e)**2/(2*scale**2))
        R = np.kron(R_matrix(-np.pi/4, np.pi/2, np.pi/2), R_matrix(np.pi/2(
        I += XI_matrix(R)*p*de

    return I

e = 0.5
np.trace(Integrate(e_min = -e, e_max = e, N = 100))
```

Out[46]:

```
(3.999998257107002-6.018531076210112e-36j)
```

In [127…

```
# Эволюция через операторы Краусса

def Kraus_eval(Kraus_list, rho0):
    rho = 0
    for K in Kraus_list:
        K = np.reshape(K, (4,4))
        rho += K @ rho0 @ K.T.conjugate()
    return rho
```

In [128…
```
# Посчитаем усреднённую Xi-матрицу по шуму и сравним с точной

e = 0.8
np.sum(XI_matrix(np.kron(R_matrix(-np.pi/4, np.pi/2, np.pi/2), R_matrix(np
```

Out[128…
```
(0.000832341526704569-3.4580070709983744e-19j)
```

In [129…
```
# Не совпадает с точным значением

np.sum(XI_matrix(np.kron(R_matrix(-np.pi/4, np.pi/2, np.pi/2), R_matrix(np
```

Out[129…
```
(-25.18488187841352-8.651933336434325e-17j)
```

In [164…
```
# Посчитаем Xi-матрицу, усреднённую по заддному распределению шума

e = 0.9
XI = Integrate(e_min = -e, e_max = e, N = 10000, loc = 0, scale = 0.1)
u, s, vh = LA.svd(XI)
```

In [165…
```
# Получим операторы краусса в виде столбцов матрицы

sm = np.zeros((len(s), len(s)))
for i in range(len(s)):
    sm[i][i] = s[i]

Kraus_list = np.dot(u,np.sqrt(sm)).T
```

In [132…
```
# Получим распределения вероятностей из пункта 3 для случаев с шумом и без

zero_vec = np.zeros(2**d)
zero_vec[0] = 1

res_list = []
res_list_noisy = []
state = Qstate(d)
state.build_pure_random_state()
rho0 = state.get_rho()

rho = Kraus_eval(Kraus_list, rho0)
result_F = np.dot(np.dot(zero_vec, rho),zero_vec)

L = 10000
start_time = time.time()
for i in range(L):
    res_list_noisy.append(get_prob(state, loc = 0, scale = 0.1))
for i in range(L):
    res_list.append(get_prob(state, noisy = False))

t = time.time() - start_time
t
```

Out[132…
```
3514.700817346573
```

In [166…

```
# Получим аналогичную вероятность с помощью усреднённой Xi-матрицы

rho = Kraus_eval(Kraus_list, rho0)
result_F = np.dot(np.dot(zero_vec, rho),zero_vec.conj())
result_F
```

Out[166…

```
(0.34438909721954963-0.5174986333916715j)
```

In [174…

```
# Построим распределения

# plt.figsize(8,8)
plt.title('Распределение вероятностей получения результата 1')
plt.hist(res_list_noisy, bins = 100, label = 'noisy', alpha = 0.3)
plt.hist(res_list, bins = 100, label = 'clear', alpha = 0.3)
plt.vlines(x = np.abs(result_F), ymin = 0, ymax = L/20, label = 'Kraus eva
plt.legend()
plt.show()
```

Часть 2:

In [118…

```
from Gates import *
from functions import *
from state import *
```

In [119…

```
# Зададим нулевой вектор

n = 6
state = Qpsi(n)
state.build_zero_state()
```

In [120…

```
state.get_coefs()
```

Out[120…

```
array([1.+0.j, 0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j,
       0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j,
       0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j,
       0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j,
       0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j,
       0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j,
```

```
       0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j,
       0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j])
```

In [121…

```
# Фурье преобразование

def Fourier_transfer(state, n, noisy = False, e = 10e-6):
#     state = Qpsi(n)
#     state.build_zero_state()

    for i in range(n):
        state.apply_U(H(), axis = [i])
        if noisy == True:
            eps = np.random.normal(0, e)
            state.apply_U(R_matrix(eps*np.pi, np.pi/2, eps), axis = [i])
        k = 2
        for j in range(i+1, n):
            state.apply_U(CR(k = k), axis = [j,i])
            k +=1
    return state.get_coefs()
```

In [122…

```
# Расчет метрики F

def calc_F(a,b):
    return np.dot(a,b)**2/(np.dot(a,a)*np.dot(b,b))
```

In [123…

```
calc_F([0.9,1], [1,0.9])
```

Out[123…

```
0.98898080034187
```

In [124…

```
# Произведём Фурье преобразование по 1000 раз для различных уровней шума и

L = 1000

F_list = []
lql = []
hql = []

state = Qpsi(n)
state.build_random_state()

coefs = state.get_coefs()
coefs_clear = Fourier_transfer(state, n, noisy = False)
e_list = [0, 0.0001, 0.0005, 0.001, 0.005, 0.01, 0.05, 0.1]
st = time.time()

for e in e_list:
    F_mean = []
    for i in range(L):
        state.set_coefs(coefs)
        coefs_noisy = Fourier_transfer(state, n, noisy = True, e = e)
        print(time.time() - st)

        F = calc_F(coefs_clear, coefs_noisy)
        F_mean.append(F)
    low_q = np.quantile(F_mean, 0.25)
    high_q = np.quantile(F_mean, 0.75)

    lql.append(low_q)
    hql.append(high_q)
    F_list.append(np.mean(F_mean))
```

```
0.012974739074707031
0.01965808868408203
0.026621580123901367
0.03351283073425293
0.040342092514038086
0.047020912170410156
0.053802490234375
0.060545921325683594
0.06718969345092773
0.07458877563476562
0.08250212669372559
0.09054112434387207
0.09731340408325195
0.10450530052185059
0.11103963851928711
0.11745953559875488
0.12379717826843262
0.13034272193908691
0.13669061660766602
0.14310884475708008
0.14962267875671387
0.15596532821655273
0.1621873378753662
0.16860318183898926
0.17494893074035645
0.1813652515411377
0.18795990943908691
0.19440865516662598
```

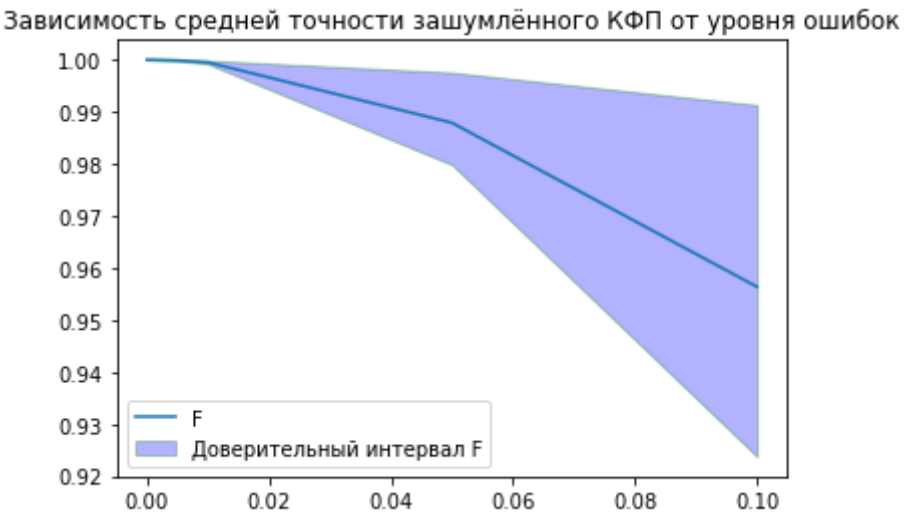
```
53.49189615249634
53.4984233379364
53.50466322898865
53.511295795440674
53.517966747283936
53.52491903305054
53.5316641330719
53.538344383239746
53.54523324966431
53.551958084106445
53.558594942092896
```

In [125…

```
# построим зависимость средней точности Квантового Фурье преобразования в 

plt.title('Зависимость средней точности зашумлённого КФП от уровня ошибок'
plt.plot(e_list, F_list, label = 'F')
plt.fill_between(e_list, lql, hql, edgecolor='g', facecolor='b', alpha=0.3
plt.legend()
plt.show()
```

```
/home/stas/anaconda3/lib/python3.7/site-packages/matplotlib/axes/_axes.py:5
359: ComplexWarning: Casting complex values to real discards the imaginary
part
  pts[0] = start
/home/stas/anaconda3/lib/python3.7/site-packages/matplotlib/axes/_axes.py:5
360: ComplexWarning: Casting complex values to real discards the imaginary
part
  pts[N + 1] = end
/home/stas/anaconda3/lib/python3.7/site-packages/matplotlib/axes/_axes.py:5
363: ComplexWarning: Casting complex values to real discards the imaginary
part
  pts[1:N+1, 1] = dep1slice
/home/stas/anaconda3/lib/python3.7/site-packages/matplotlib/axes/_axes.py:5
365: ComplexWarning: Casting complex values to real discards the imaginary
part
  pts[N+2:, 1] = dep2slice[::-1]
```



In [ ]: