# Rust Quantum Library

The quantum library in Rust contains several useful measures of quantum entanglement that are calculated from the density matrix representing a quantum state. This library is generalized for any density matrix with dimension of $2^n$. The Rust library uses ndarray and the ndarray_linalg packages.

We can run Rust from a Jupyter notebook by installing EvCxR Jupyter kernel. The Jupyter notebook runs with the **plotters** crate. Setup can be found from https://datacrayon.com/posts/programming/rust-notebooks/setup-anaconda-jupyter-and-rust. Miniconda is not necessary, Anaconda will work the same. Be careful when installing the actual kernel in the tutorial (shown below) - it is important to check on **"other installation methods"** in order for proper installation for Linux, Windows, and Mac.

## Install the EvCxR Jupyter Kernel

Now we'll install the EvCxR Jupyter Kernel. If you're wondering how it's pronounced, it's been mentioned to be "Evic-ser". This is what will allow us to execute Rust code in a Jupyter Notebook.

You can get other installation methods methods for EvCxR if you need then, but we will be using:

```
cargo install evcxr_jupyter --version 0.5.3
evcxr_jupyter --install
```

Below are the various quantum functions along with the functions that perform linear algebra operations.

# 1 Quantum functions

## 1.1 Create a density matrix $\rho$ (*create_density_matrix*)

**Input:** The wavefunction expressed as a column vector of coefficients

**Input type:** pub type VecC64 = ndarray::Array1<c64>

**Input Example:** For the Bell state $\phi_+ = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$,

let norm_const = 1./2_f64.sqrt();

let bell_phi_plus = array![c64::new(norm_const , 0.0) , c64::new(0.0 , 0.0) , c64::new(0.0 , 0.0) , c64::new(norm_const , 0.0)];

**Output:** The density matrix, which is square and Hermitian.

**Output type:** pub type MatrixC64 = ndarray::Array2<c64>

**Output Example:** dens matrix for phi_+ =

[[0.4999999999999999+0i, 0+0i, 0+0i, 0.4999999999999999+0i],

[0+0i, 0+0i, 0+0i, 0+0i],

[0+0i, 0+0i, 0+0i, 0+0i],

[0.4999999999999999+0i, 0+0i, 0+0i, 0.4999999999999999+0i]]

The density matrix is a more general way of representing the state of a quantum system. It is represented as $\rho = |\psi\rangle \langle\psi|$, where the wavefunction $|\psi\rangle$ is represented as a column vector.
With this same example, the function performs the operation

$$\rho = |\phi_+\rangle \langle\phi_+| = \frac{1}{2}\begin{pmatrix}1\\0\\0\\1\end{pmatrix}\begin{pmatrix}1 & 0 & 0 & 1\end{pmatrix}^* = \frac{1}{2}\begin{pmatrix}1 & 0 & 0 & 1\\0 & 0 & 0 & 0\\0 & 0 & 0 & 0\\1 & 0 & 0 & 1\end{pmatrix}$$

### 1.2 Find purity (*find_purity*)

**Input**: $\rho$ (Density Matrix)

**Input Type:** pub type MatrixC64 = ndarray::Array2<c64>

**Input Example:**

array![ [c64::new(0.5 , 0.) , c64::new(0. , 0.) ],

[c64::new(0. , 0.) , c64::new(0.5 , 0.)] ]

**Output:** Float

**Output Type:** f64

**Output Example:** 0.5

Simply returns the trace of $\rho^2$ as a f64 type. For an idea on where this purity lies within the range of possible values, print a statement such as println!("The purity lies between {} and 1", 1./( find_dim(rho_sqrd) ) ). The *find_dim* function finds the dimension of the square matrix. For example, a 4 x 4 matrix has dim = 4.

### 1.3 Find fidelity (*find_fidelity*)

**Input:** Two density matrices $\rho_1$ and $\rho_2$

**Input Type:** pub type MatrixC64 = ndarray::Array2<c64>

**Input Example:** let rho_1 = array![

[c64::new(0.25 , 0.0) , c64::new(0.0 , 0.0) , c64::new(0.0 , 0.0) , c64::new(0.0 , 0.0) ],

[c64::new(0.0 , 0.0) , c64::new(0.25 , 0.0) , c64::new(0.0 , 0.0) , c64::new(0.0 , 0.0) ],

[c64::new(0.0 , 0.0) , c64::new(0.0 , 0.0) , c64::new(0.25 , 0.0) , c64::new(0.0 , 0.0) ],

[c64::new(0.0 , 0.0) , c64::new(0.0 , 0.0) , c64::new(0.0 , 0.0) , c64::new(0.25 , 0.0)]

];

let rho_2 = rho_1;

**Output:** A number between 0 and 1

**Output Type:** f64

**Output Example:** 1

This function is a distance measurement of two quantum states $\rho_1$ and $\rho_2$. It is expressed as $F = tr\sqrt{\sqrt{\rho_1}\rho_2\sqrt{\rho_1}}$. This function calls *find_sqr_root_of_matrix* to calculate $\sqrt{\rho_1}$ and the overall square root of $\sqrt{\rho_1}\rho_2\sqrt{\rho_1}$.

### 1.4 Find concurrence (*find_concurrence*)

**Input:** The density matrix $\rho$

**Input Type:** pub type MatrixC64 = ndarray::Array2<c64>

**Input Example:** The state $\cos(\theta)\left|00\right\rangle + \sin(\theta)\left|11\right\rangle$ for $\theta = 30°$ is represented by the vector:

```
pub const THETA: f64 = PI*(30./180.);
let psi_part_entangled: VecC64 = array![c64::new(THETA.cos(), 0.0), c64::new(0.0, 0.0), c64::new(0.0, 0.0), c64::new(THETA.sin(), 0.0)];
```

**Output:** A number between 0 and 1

**Output Type:** f64

**Output Example:** 0.866 (which is the rounded from $\sqrt{3}/2$)

For two qubits, concurrence $C(\rho) = max(0, \lambda_1 - \lambda_2 - \lambda_3 - \lambda_4)$, where $\lambda_1, ...\lambda_4$ are the eigenvalues in decreasing order (ie $\lambda_1$ is the highest eigenvalue) of the matrix $R = \sqrt{\sqrt{\rho}\tilde{\rho}\sqrt{\rho}}$.

In this case, $\tilde{\rho} = (\sigma_y \otimes \sigma_y)\rho^*(\sigma_y \otimes \sigma_y)$. $\rho^*$ is the complex conjugate of $\rho$, and $\sigma_y$ is the Pauli-y spin matrix.

This function has been extended from 2 qubits (4 x 4 density matrix) to 3 or more qubits. The function will give wrong answers for odd numbers of qubits.

The extension uses $C(\rho) = max(0, \lambda_1 - \sum \lambda_i)$, where $i$ goes from 2 to the density matrix dimension. The dimension of the density matrix is $2^n$, where $n$ is the number of qubits. Also in this case $\tilde{\rho} = (\sigma_y \otimes^n \sigma_y)\rho^*(\sigma_y \otimes^n \sigma_y)$.

The tensor product operation for $\otimes^n$ is called using the *find_tensor_product* function, and uses an iterator with the fold method to perform the tensor product $n$ times.

### 1.5 Find trace norm (*find_trace_norm*)

**Input:** The density matrix $\rho$

**Input Type:** pub type MatrixC64 = ndarray::Array2<c64>

**Input Example:**

For the Bell state $\phi_+ = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$,

let norm_const = 1./2_f64.sqrt();

let bell_phi_plus = array![c64::new(norm_const , 0.0) , c64::new(0.0 , 0.0) , c64::new(0.0 , 0.0) , c64::new(norm_const , 0.0)];

**Output:** A float from 0 to 1

**Output Type:** f64

**Output Example:** 1

The trace norm is expressed as $||\rho||_1 = tr\sqrt{\rho^\dagger \rho}$. $\rho^\dagger$ is the complex conjugate and transpose of the matrix $\rho$. The function calls *find_sqr_root_of_matrix*$(\rho^\dagger \rho)$. For Hermitian, normalized density matrices, the output is always 1.

### 1.6 Find negativity (*find_negativity*)

**Input:** The density matrix $\rho$

**Input Type:** pub type MatrixC64 = ndarray::Array2<c64>

**Input Example:** For a 5 qubit maximally mixed density matrix, let rho_mixed_diag: VecC64 = Array::from_elem( 32, c64::new(1./32., 0.0) );

let rho_mixed: MatrixC64 = MatrixC64::from_diag(rho_mixed_diag);

**Output:** A number between 0 and 1

**Output Type:** f64

**Output Example:** 0.0

Th negativity can be expressed as $N = \frac{||\rho^{\Gamma_A}||_1 - 1}{2}$ where $\rho^{\Gamma_A}$ is the partial transpose of a substate of $\rho$. $||\rho^{\Gamma_A}||_1$ is the trace norm of that partial transpose. Although the trace norm for density matrices is 1, the trace norm of the partially transposed matrix can vary from 0 to 1. The Peres-Horodecki criterion separates $\rho$ into the tensor product of two states A and B. $\rho^{\Gamma_A}$ is called with the function *find_partial_transpose*$(\rho)$.

## 1.7 Find log negativity (*find_log_negativity*)

**Input:** The density matrix $\rho$

**Input Type:** pub type MatrixC64 = ndarray::Array2<c64>

**Input Example:** For the Bell State $\phi_- = \frac{1}{\sqrt{2}}(|00\rangle - |11\rangle)$:

pub const NORM_CONST: f64 = 1./std::f64::consts::SQRT_2;

let bell_phi_minus_vec: VecC64 = array![ c64::new(NORM_CONST , 0.0) , c64::new(0.0 , 0.0) , c64::new(0.0 , 0.0) , c64::new(-NORM_CONST , 0.0) ];

let rho_bell_psi_plus = create_density_matrix(BELL_PSI_PLUS_VEC);

**Output:** A number between 0 and 1

**Output Type:** f64

**Output Example:** 0.0

This function lets $N = find\_negativity(\rho)$. Then it takes $log_2(2N + 1)$.

### 1.8 Find Schmidt number (*find_schmidt_number*)

**Input:** The joint spectral intensity (JSI) matrix

**Input Type:** pub type MatrixF64 = ndarray::Array2<f64>

**Input Example:** A 200 x 200 JSI (located inside the examples folder)

**Output:** An entanglement measurement of the JSI.

**Output Type:** f64

**Output Example:** 6.74 (rounded)

This computation is done in several steps:

1) Take the square root of the elements of the JSI (joint spectral intensity) to find the JSA (joint spectral amplitude). The JSI is a matrix of floats from SPDCalc.

2) Find the singular value decomposition (SVD) of the JSA. A SVD calculation decomposes the into three matrices, $USV^T$.

3) A normalization constant is found by taking the inverse of the sum of the squares of the singular values of the S matrix (eigenvalues), i.e. $A = 1/\sum \lambda_i^2$.

4) The S matrix is renormalized by multiplying the $\sqrt{A} * S$.

5) From the normalized S matrix, take the sum of the eigenvalues to the fourth power, i.e. $\sum \lambda_{i,norm}^4$. The Schmidt number is $k = 1/\sum \lambda_{i,norm}^4$.

### 1.9 Create two source Hong-Ou-Mandel graph (*find_two_source_hom*)

**Inputs:** A signal vector of frequency modes $\omega$, an idler vector of modes $\omega$, a joint spectral amplitude (JSA) of modes $\omega$, and a vector of time intervals dt in femtoseconds.

**Input Types:** signal: VecF64, idler: VecF64, jsa: MatrixC64, dt: VecF64

**Output:** A graph of time intervals (dt) on the x axis and coincidence probabiliities on the y axis.

**Output Example** An example is located in the "two_source_hom_plot.ipynb" under qm/examples.

When calculating the two-source Hong-Ou-Mandel graph, we consider four photons from two identical crystals, i.e. two signal photons and two idler photons. The relevant integral to compute is

$$\int_{\omega_s}\int_{\omega_i}\int_{\omega_s'}\int_{\omega_i'} \psi(\omega_s, \omega_i, \omega_s',\ \omega_i') - e^{i\phi}\psi(\omega_s', \omega_i, \omega_s,\ \omega_i')\ d\omega_i'\ d\omega_s'\ d\omega_i\ d\omega_s.$$

The wave function $\psi$ is equivalent to the joint spectral amplitude (JSA) for all four photons. Since the photons come from two separate sources, this joint amplitude is separable into two parts, so that

$$\psi(\omega_s, \omega_i, \omega_s^{'}, {\omega_i}') = JSA_1(\omega_s, \omega_i) \otimes JSA_2(\omega_s^{'}, {\omega_i}')$$

.Since the two sources are identical, we conclude that $JSA_1 = JSA_2$. So we only need to compute the JSA once and then simply reference different indices for the different signal and idler photons. Using this notation, the integrand becomes

$$JSA(\omega_s, \omega_i) \otimes JSA(\omega_s^{'}, \omega_i^{'}) - e^{i\phi} JSA(\omega_s^{'}, \omega_i) \otimes JSA(\omega_s, \omega_i^{'})$$

.

We can simply iterate over many sums rather than explicitly calculating this integral. The variable *phi* depends on time and wavelength difference between two photons of interest. For signal-signal interference, $\phi_{ss} = 2\pi\Delta t(\frac{1}{\lambda_s} - \frac{1}{\lambda_i})$. For idler-idler interference, $\phi_{ii} = 2\pi\Delta t(\frac{1}{\lambda_s'} - \frac{1}{\lambda_i'})$. For signal-idler interference, $\phi_{si} = 2\pi\Delta t(\frac{1}{\lambda_s} - \frac{1}{\lambda_i'})$.

In the code, we let A, B, C, D stand for the different JSA configurations, between $\omega_s$ and $\omega_i$, $\omega_s'$ and $\omega_i'$, $\omega_s'$ and $\omega_i$, and $\omega_s$ and $\omega_i'$ respectively.

The *find_two_source_hom_norm* function normalizes the coincidence probabilities.

### 1.10 Find two source Hong-Ou-Mandel norm (*find_two_source_hom_norm*)

**Inputs:** A signal vector of frequency modes $\omega$, an idler vector of modes $\omega$, and a joint spectral amplitude (JSA) of modes $\omega$.

**Input Types:** signal: VecF64, idler: VecF64, jsa: MatrixC64

**Output:** A value that normalizes the two_source_hom for each time interval dt.

**Output Type:** f64

**Output Example:** 6.41e+61_f64 (rounded)

Normalizes the coincidence probabilities in the two source HOM calculation.

# 2 Matrix Operations

### 2.1 Find the symmetric square root of a matrix (*find_symmetric_square_root*)

**Input:** Any square, Hermitian matrix

**Input Type:** pub type MatrixC64 = ndarray::Array2<c64>

**Input Example:**

array![ [c64::new(0.0 , 0.0) , c64::new(0.0 , 0.0) , c64::new(0.0 , 0.0) ],

[c64::new(0.0 , 0.0) , c64::new(0.5 , 0.0) , c64::new(0.0 , 0.5) ],

[c64::new(0.0 , 0.0) , c64::new(0.0 , -0.5) , c64::new(0.5 , 0.0) ],];

**Output:** The square root of the matrix

**Output Type:** pub type MatrixC64 = ndarray::Array2<c64>

**Output Example:**

array![ [c64::new(0.0 , 0.0) , c64::new(0.0 , 0.0) , c64::new(0.0 , 0.0) ],

[c64::new(0.0 , 0.0) , c64::new(0.5 , 0.0) , c64::new(0.0 , 0.5) ],

[c64::new(0.0 , 0.0) , c64::new(0.0 , -0.5) , c64::new(0.5 , 0.0) ],];

Computes the symmetric square root of a Hermitian matrix. In the example above, the square root of the matrix is equivalent to the matrix itself.

*****Bug in LAPACK involving numerical precision**

For some of the entanglement measurements, the square root of the density matrix is required. Using a LAPACK function such as matrix.ssqrt(UPLO:Lower).unwrap() is the easiest and fastest route, but LAPACK has a bug and fails to compute. It's suspected that the matrix needs to be semi-positive definite (eigenvalues need to be >= 0) in order for it to work.

So, the function was written by decomposing the Hermitian matrix as $M = SDS^{-1}$, where $S$ and $S^{-1}$ are complex unitary matrices and $D$ is a real diagonal matrix. $S$ is a matrix of the eigenvectors of M, and $D$ is has the eigenvalues of M in the diagonal.Since $D$ is diagonal, one can take the square of the elements in order to find $\sqrt{D}$. So, $\sqrt{M} = S\sqrt{D}S^{-1}$.

But, one problem is that again, when doing this decomposition, the matrix $M$ needs to be semi-positive definite. It usually is, but LAPACK will sometimes find that one eigenvalue is a very small negative number, such as $2.8 \times 10^{-15}$. A rescaling function is called in order to fix this issue.

The rescaling function takes matrix M as an input and finds its eigenvalues and eigenvectors, and sets any negative eigenvalues to 0. Then, matrix $M$ can be decomposed into $M = SDS^{-1}$. The rescaling function returns ($D$, $S$) as a tuple. Then, the square root function returns $\sqrt{M} = S\sqrt{D}S^{-1}$.

Unfortunately after rescaling the function and calling LAPACK with matrix.ssqrt(UPLO:Lower).unwrap() still fails in the computation. This is why ($D$, $S$) is returned as a tuple rather than a rescaled density matrix.

## 2.2 Find matrix dimension (***find_dim***)

**Input:** Square Matrix

**Input Type:** pub type MatrixC64 = ndarray::Array2<c64>

**Input Example:**

array![ [c64::new(3., 1.) , c64::new(-1., 1.) ],

[c64::new(2., -1.) , c64::new(-2., -1.)] ]

**Output:** Integer

**Output Type:** i32

**Output Example:** 2

A simple function that calculates the length of a square matrix. For example, if a matrix is 8 x 8, this will return 8 as a type i32 (integer 32 bit).

### 2.3 Find partial transpose (*find_partial_transpose*)

**Input:** Any square matrix

**Input Type:** pub type MatrixC64 = ndarray::Array2<c64>

**Input Example:** let matrix: MatrixC64 = array![

[c64::new(1.0 , 0.0) , c64::new(2.0 , 0.0) , c64::new(3.0 , 0.0) , c64::new(4.0 , 0.0) ] ,

[c64::new(5.0 , 0.0) , c64::new(6.0 , 0.0) , c64::new(7.0 , 0.0) , c64::new(8.0 , 0.0) ] ,

[c64::new(9.0 , 0.0) , c64::new(13.0 , 0.0) , c64::new(11.0 , 0.0) , c64::new(12.0 , 0.0)] ,

[c64::new(10.0 , 0.0) , c64::new(14.0 , 0.0) , c64::new(15.0 , 0.0) , c64::new(16.0 , 0.0)] ];

**Output:** A square matrix

**Output Type:** pub type MatrixC64 = ndarray::Array2<c64>

**Output Example:**

[c64::new(1.0 , 0.0) , c64::new(2.0 , 0.0) , c64::new(3.0 , 0.0) , c64::new(7.0 , 0.0) ] ,

[c64::new(5.0 , 0.0) , c64::new(6.0 , 0.0) , c64::new(4.0 , 0.0) , c64::new(8.0 , 0.0) ] ,

[c64::new(9.0 , 0.0) , c64::new(10.0 , 0.0) , c64::new(11.0 , 0.0) , c64::new(12.0 , 0.0)] , [c64::new(13.0 , 0.0) , c64::new(14.0 , 0.0) , c64::new(15.0 , 0.0) , c64::new(16.0 , 0.0)] ];

### 2.4 Find tensor product (*find_tensor_product*)

**Input:** One square matrix with dim $m \times m$ and another with dim $n \times n$

**Input Type:** pub type MatrixC64 = ndarray::Array2<c64>

**Input Example:** let matrix: MatrixC64 = array![

[c64::new(1.0 , 0.0) , c64::new(1.0 , 0.0)] ,

[c64::new(1.0 , 0.0) , c64::new(-1.0 , 0.0)] ,

];

**Output:** A square matrix with dim $mn \times mn$

**Output Type:** pub type MatrixC64 = ndarray::Array2<c64>

**Output Example:**

[c64::new(1.0 , 0.0) , c64::new(1.0 , 0.0) , c64::new(1.0 , 0.0) , c64::new(1.0 , 0.0) ] ,

[c64::new(1.0 , 0.0) , c64::new(-1.0 , 0.0) , c64::new(1.0 , 0.0) , c64::new(-1.0 , 0.0) ] ,

[c64::new(1.0 , 0.0) , c64::new(1.0 , 0.0) , c64::new(-1.0 , 0.0) , c64::new(-1.0 , 0.0)] ,

[c64::new(1.0 , 0.0) , c64::new(-1.0 , 0.0) , c64::new(-1.0 , 0.0) , c64::new(1.0 , 0.0)] ];