QI Mathematica Package – Brief Manual

Roger Colbeck[*]
Department of Mathematics, University of York, YO10 5DD, UK
(Dated: 29th January 2024)

The following is a brief description of a selection of Mathematica commands that may be of use to people studying quantum information. I make no guarantee that they are well written, free of bugs, optimized for speed and memory usage, etc. Most are functions that I have written because I have found them useful, and I have used them and tested them to a reasonable extent (i.e., until I have become confident in them, at least in the instances in which I have employed them). If you find any bugs, I would be pleased to hear of them. As far as I am aware, all commands work in Mathematica 12.

I. INSTALLATION

No special installation is needed. The file QI.m can be downloaded from `https://github.com/rogercolbeck/QI`. Simply open the file and press Run All Code. However, if you wish to automatically load these functions on starting Mathematica:

1. Change to your autoload directory (usually the Autoload sub-directory of your user addons directory, found using the command $UserAddOnsDirectory).
2. Create a subdirectory QI.
3. Put the file QI.m in this directory.
4. Rename the file init.m.

A file, `https://www-users.york.ac.uk/~rc973/QI/QI_examples.nb` collects some simple examples illustrating the use of these functions. A palette, `https://www-users.york.ac.uk/~rc973/QI/Palette.nb` can also be downloaded which contains some useful symbols (including $\otimes$).

II. GENERAL SYNTAX

In order to use these commands, all vectors must be entered as matrices (this makes them of the right "shape" that the same commands (such as tensor product) can be used for both vectors and matrices without modification). That is, to assign the vector $\begin{pmatrix} 0 \\ 1 \end{pmatrix}$ to v one would enter v = {{0}, {1}}. It is easy to convert a standard Mathematica vector (i.e., one entered as v = {0, 1}, which I call a list) to this form by using v = Transpose[{v}].

The error handling of these commands is basic (although some checks are done, mostly any displayed errors come out of Mathematica's built-in functions).

Many of the commands use Chop for comparisons of numerical values. By default, Chop treats numbers smaller than 10^{-10} as 0. This means that some commands may do the wrong thing with numerical inputs that are smaller than 10^{-10}.

III. COMMAND LIST

A. Manipulating Quantum States

CT[M]: Equivalent to ConjugateTranspose[M].

[*]Electronic address: roger.colbeck@york.ac.uk

KetV[i, d]: This creates the i^{th} basis vector in dimension d, where $i \in \{0, \ldots, d-1\}$.

BraV[i, d]: This creates the i^{th} dual basis vector in dimension d, where $i \in \{0, \ldots, d-1\}$. BraV[i, d] is equivalent to CT[KetV[i, d]].

DM[vec]: Converts the vector vec to a density matrix.

CircleTimes[A, B] (⊗): This performs the tensor product and can be entered as either A ⊗ B or CircleTimes[A, B]. The ⊗ symbol can be found in the operators→general section of the complete characters palette, or may be entered using the keyboard sequence ESCAPE, c , *, ESCAPE.

Tensor[A, B, pos, desc]: This perform the tensor product of A and B, placing the systems in the positions specified by pos (a string of 1s and 2s) where the systems have dimensionalities specified by desc (note that desc does NOT specify the dimensionality of the matrices A and B). E.g. Tensor[A, B, $\{1, 2, 1, 2\}, \{\{a_1, a_2\}, \{b_1, b_2\}, \{c_1, c_2\}, \{d_1, d_2\}\}$] requires A to be the tensor product of a $a_1 \times a_2$ matrix with a $c_1 \times c_2$ matrix, i.e., to have dimension $a_1c_1 \times a_2c_2$, and B to be the tensor product of a $b_1 \times b_2$ matrix with a $d_1 \times d_2$ matrix. The output has A on the 1st and 3rd systems and B on the 2nd and 4th systems. If desc is a simple list, e.g., $\{a_1, b_1, c_1, d_1\}$ then A and B must be either be square matrices of size a_1c_1 and b_1d_1 or vectors of size a_1c_1 and b_1d_1.

TensorPower[M, n]: This raises matrix M to the n-th tensor power, i.e., computes $M^{\otimes n}$.

ExchangeSystems[A, newpos, desc]: This takes an operator representing systems in one order and converts it to another, as specified by newpos. The first entry of newpos gives the position that the first system moves to, etc. A can be a vector or a matrix. desc should be a list of the dimensions, e.g., desc=$\{\{2, 4\}, \{3, 5\}\}$ means that A is considered as the tensor product of a 2×3 matrix with a 4×5 matrix (so A should have dimensions 8×15). If the subsystems are square, or are vectors, desc need only be the number of rows. For instance, ExchangeSystems[v1⊗v2⊗v3,{3,1,2},{2,3,4}] gives v2⊗v3⊗v1, where v1 has dimension 2×1, v2 has dimension 3×1 and v3 has dimension 4×1. [Or v1 has dimension 2×2, v2 has dimension 3×3 and v3 has dimension 4×4.]

One can also do ExchangeSystems[DM[v1⊗v2⊗v3],{3,1,2},{2,3,4}], which is the same as DM[v2⊗v3⊗v1].

DirectSum[A,B,C,...]: This computes the direct sum of the square matrices A, B, C, ..., i.e. A ⊕ B ⊕ C ⊕ ...

QubitPartialTrace[M, {i, j, ...}]: This traces out the i, j, ... qubits of M, assuming that M is the tensor product of multiple qubit spaces.

PartialTrace[M, dim_1, dim_2, tr]: This traces out the $\text{tr} \in \{1, 2\}$ subsystem of M, where M has the Hilbert space structure $\mathcal{H}_1 \otimes \mathcal{H}_2$, where $\mathcal{H}_1$ is dim_1-dimensional and $\mathcal{H}_2$ is dim_2-dimensional.

PT[M, keep, desc]: This traces out the systems specified by keep of M, where the dimensions of the subsystems in M are specified by desc. The argument keep should be a binary vector where 0s are systems to be traced out and 1s are systems to keep. The description desc should be a list of the dimensions of the subsystems e.g. keep=$\{1, 0, 1, 0\}$, desc=$\{2, 4, 2, 3\}$ traces out the 2nd and 4th systems of M, where M is an operator on $\mathbb{C}^2 \otimes \mathbb{C}^4 \otimes \mathbb{C}^2 \otimes \mathbb{C}^3$ (i.e. a 48×48 matrix). The output is a 4×4 matrix.

PTrans[M, action, desc]: This takes the partial transpose of the systems specified by action of M, where the dimensions of the subsystems in M are specified by desc. The argument action should be a binary vector where 1s are systems to be transposed and 0s are systems to do nothing to. The description desc should be a list of the dimensions of the subsystems e.g. action=$\{1, 0, 1, 0\}$, desc=$\{2, 4, 5, 3\}$ transposes the 1st and 3rd systems of M, where M is an operator on $\mathbb{C}^2 \otimes \mathbb{C}^4 \otimes \mathbb{C}^5 \otimes \mathbb{C}^3$ (i.e. a 120×120 matrix).

BasisForm(S)[vec, desc]: This writes out a vector, vec, with structure given by desc in the computational basis, giving one line for each non-zero component (the BasisFormS version gives the output all on one line) e.g. BasisFormS[{1,0,0,1},{2,2}] returns $+1|0, 0\rangle + 1|1, 1\rangle$ i.e. it is the state $|00\rangle + |11\rangle$, while BasisFormS[{1,0,0,1},{4}] returns $+1|0\rangle + 1|3\rangle$. Note: this command applies Flatten to vec, so will do the wrong thing with matrix inputs.

Purify[rho]: This gives a state that purifies rho (outputting as a vector). The chosen purification takes a copy of the original eigenvector.

SchmidtDecomposition[v, desc]: This gives the Schmidt decomposition of the state v where desc is a two component list of the dimensions across which the decomposition is taken. E.g., if v has eight dimensions, then desc can be $\{2, 4\}$ or $\{4, 2\}$. The output is in the form $\{\mathsf{c}, \mathsf{vecs}_1, \mathsf{vecs}_2\}$, where c is a list of Schmidt coefficients, vecs_1 are the Schmidt vectors on system 1, and vecs_2 are those on system 2 (in the same order as the coefficients in c). In other words, $\mathsf{Sum}[\mathsf{c}[[\mathsf{i}]] * \mathsf{vecs}_1[[\mathsf{i}]] \otimes \mathsf{vecs}_2[[\mathsf{i}]], \{\mathsf{i}, 1, \mathsf{Min}[\mathsf{desc}]\}]$ is equal to v.

DiagonalizingUnitary[M]: For Normal matrix M, (i.e., $\mathsf{MM}^\dagger = \mathsf{M}^\dagger\mathsf{M}$) this returns $\{\mathsf{U}, \mathsf{D}\}$, where $\mathsf{M} = \mathsf{UDU}^\dagger$, U is unitary, and D is diagonal.

EigensystemExact[A, (prec)]: For matrix A, this returns $\{\mathsf{vals}, \mathsf{vecs}\}$, where vals are the eigenvectors and vecs are the corresponding eigenvectors, ensuring that the latter are orthonormal for exact inputs (Mathematica's Eigensystem does not do this by default). Note that doing so can be slow. Note also that this command may not behave correctly with symbolic inputs. The eigenvalues are sorted using private function OrderingF, hence by the size of their real part, followed by the size of their imaginary part (taking these parts to be equal if within prec of each other). If all the eigenvalues are real, they are sorted from smallest to largest, for example. The optional argument prec can be used to specify the tolerance for deciding whether eigenvalues are equal (by default this is set to that used by Chop[]).

SimultaneouslyDiagonalize[A, B, (prec)]: For commuting normal matrices A, B, this returns U such that U is unitary and $\mathsf{U}^\dagger\mathsf{AU}$ and $\mathsf{U}^\dagger\mathsf{BU}$ are diagonal. This has an optional third argument, which is the precision to which two numerical values are assumed to be equal. By default, this is the standard for Chop[], i.e., 10^{-10}.

BlochSphere[ρ]: This gives the point on the Bloch sphere specified by ρ, in the form of the vector $\{\mathsf{r}_1, \mathsf{r}_2, \mathsf{r}_3\}$ satisfying $\rho = \frac{1}{2}\left(\mathbb{1} + \mathsf{r}_1\sigma_1 + \mathsf{r}_2\sigma_2 + \mathsf{r}_3\sigma_3\right)$.

FromBlochSphere[r]: Gives the density matrix corresponding to this point on the Bloch Sphere (see BlochSphere).

AppendCols[U]: Takes an isometry U and returns a unitary matrix by completing the columns.

FillZero[M]: Takes a matrix with more columns that rows and makes it square by adding rows of zeros.

MeasureBasis[ρ, basis]: Performs the quantum channel on ρ equivalent to measuring in the basis specified by basis then forgetting the outcome [to remove the system, use MeasurePOVM]. The input basis is expressed as a list of vectors, e.g. basis= $\{\{1, 1\}, \{1, -1\}\}/\sqrt{2}$ measures in the σ_x basis. Note that this command can be used to measure the A system of a matrix ρ_{AB} if the dimensions of ρ and basis do not match. In other words, if the basis vectors are $\{v_i\}$, this computes $\sum_i |v_i\rangle\langle v_i|\rho|v_i\rangle\langle v_i|$ (or $\sum_i(|v_i\rangle\langle v_i| \otimes \mathbb{1}_B)\rho_{AB}(|v_i\rangle\langle v_i| \otimes \mathbb{1}_B)$).

MeasureBasis[ρ, basis, sys, desc]: Performs the quantum channel on ρ equivalent to measuring in the basis specified by basis on the sys part of ρ which is partitioned as specified in desc then forgetting the outcome [to remove the system, use MeasurePOVM]. The input basis is expressed as a list of vectors, e.g. basis= $\{\{1, 1\}, \{1, -1\}\}/\sqrt{2}$ measures in the σ_x basis. The input sys should be given as a vector of 1s and 2s, where 1 is the system to measure. If the third and fourth arguments are blank, the default is to measure the A system of a matrix ρ_{AB}, where A is assumed to have the same dimensions as the specified basis.

MeasurePOVM[ρ, POVM, sys, desc]: Performs the quantum channel on ρ equivalent to measuring the POVM specified by POVM on the sys part of ρ which is partitioned as specified in desc then tracing out the measured system. The input POVM is expressed as a list of matrices (the POVM elements), e.g. POVM= $\{\{\{1, 0\}, \{0, 0\}\}, \{\{0, 0\}, \{0, 1\}\}\}$ measures in the σ_z basis. The input sys should be given as a vector consisting of 2s and one 1, where the 1 corresponds to the system to measure. For a bipartite matrix with sys= $\{1, 2\}$ and desc= $\{2, 2\}$, this computes $\sum_x |x\rangle\langle x| \otimes \mathrm{tr}_A((E_x \otimes \mathbb{1})\rho_{AB}))$.

POVMIsometry[ρ, POVM, sys, desc]: Performs the isometry on the sys part of ρ equivalent to $\sum_x |x\rangle \otimes |x\rangle \otimes \sqrt{E_x}$. The input sys should be given as a vector consisting of 2s and one 1, where the 1 corresponds to the system to measure. Tracing out the first part of the result together with the part of sys corresponding to a 1, is the same as MeasurePOVM[ρ, POVM, sys, desc].

ChoiState[channel]: Computes the Choi state of the channel channel, which should be specified as a list $\{\mathsf{K}_1, \mathsf{K}_2, \ldots, \mathsf{K}_n\}$ of Kraus operators (all that is needed is that each K_i is a matrix of the same shape). The output is $\sum_i(\mathbb{1} \otimes \mathsf{K}_i)|\gamma\rangle\langle\gamma|(\mathbb{1} \otimes \mathsf{K}_i)$, where $|\gamma\rangle = \sum_j |j\rangle \otimes |j\rangle$.

ChoiState[set1, set2]: Computes the Choi state of the linear map defined by set1 and set2, which should be specified as lists $\{\mathsf{K}_1, \mathsf{K}_2, \ldots, \mathsf{K}_\mathsf{n}\}$ and $\{\mathsf{J}_1, \mathsf{J}_2, \ldots, \mathsf{J}_\mathsf{n}\}$ (all that is needed is that each K_i and J_i is a matrix of the same shape and that there are the same number of entries of each). The output is $\sum_i (\mathbb{1} \otimes \mathsf{K}_i) |\gamma\rangle\langle\gamma| (\mathbb{1} \otimes \mathsf{J}_i^\dagger)$, where $|\gamma\rangle = \sum_j |j\rangle \otimes |j\rangle$.

ChoiChannel[state, $\mathsf{d_A}$, $\mathsf{d_B}$]: Computes the Choi channel for the Choi state state. The output is {set1, set2}, where set1 and set2 are lists $\{\mathsf{K}_1, \mathsf{K}_2, \ldots, \mathsf{K}_\mathsf{t}\}$ and $\{\mathsf{J}_1, \mathsf{J}_2, \ldots, \mathsf{J}_\mathsf{t}\}$, where t is at most $\mathsf{d_A d_B}$ and each K_i and J_i is a $\mathsf{d_B} \times \mathsf{d_A}$ matrix (i.e., the channel will be from A to B), such that the corresponding channel is $\rho \mapsto \sum_i \mathsf{K}_i \rho \mathsf{J}_i^\dagger$. If state is positive, the output has $\mathsf{J}_i = \mathsf{K}_i$ for all i.

ChannelCompress[channel]: Takes a channel specified by a list of Kraus operators and outputs another representation of the same channel, potentially with fewer Kraus operators. For a channel from dimension $\mathsf{d_A}$ to $\mathsf{d_B}$, the output has at most $\mathsf{d_A d_B}$ operators.

ExtremeChannelQ[channel, (tol)]: Checks whether the channel channel is extreme or not. Note that this involves computing the rank, so numerical instability can lead to erroneous results. Optional argument tol gives the tolerance for deciding when a singular value is zero.

B. Random Sampling

For many of these commands, adding R to the front restricts to reals, and adding F restricts to reals and chooses exact values. The fractional (F) version requires an additional argument prec which must be an integer (prec should not be used for the others). This corresponds to the maximum denominator of the fractions chosen, prior to normalization.

(R/F)PickRandomPsi[n (,prec)]: Gives a random n-dimensional normalized vector. (R) for real version, (F) for fractional (real) version. This command samples by picking uniformly distributed random numbers between 0 and 1 for the magnitude, and uniformly distributed random numbers between 0 and 2π for the phase and then normalizing the generated list. F version samples in a different way.

(R/F)PickRandomPsi2[n (,prec)]: Gives a random n-dimensional normalized vector. (R) for real version, (F) for fractional (real) version. This command samples by picking a random number between 0 and 1, and square rooting it for the first probability amplitude. It then picks between 0 and the maximum remaining weight, etc. This hence gives states with more coefficients closer to 0 than the first sampling method. The first coefficients of these states also tend to be the largest. F version samples in a different way.

(R)PickRandomPsiHaar[n]: Gives a random n-dimensional normalized vector with Haar distribution. (R) takes the absolute value of each component.

(R/F)PickRandomUnitary[n (,prec)]: Gives a random $\mathsf{n} \times \mathsf{n}$ unitary matrix. The algorithm works by picking n linearly independent vectors using PickRandomPsi and performing a Gram-Schmidt orthogonalization. (R) for real version, (F) for fractional (real) version.

(R)PickRandomUnitaryHaar[n]: Gives a random $\mathsf{n} \times \mathsf{n}$ unitary matrix with Haar distribution. The algorithm works by picking a matrix of complex numbers each with normally distributed real and imaginary parts and performing Gram-Schmidt orthogonalization. (R) for real version.

(R/F)PickRandomIsometry[dim_1, dim_2 (,prec)]: Gives a random isometry from dim_1 to dim_2. (R) for real version, (F) for fractional (real) version.

(R/F)PickRandomRho[n (,prec) (,rank)]: Gives a random $\mathsf{n} \times \mathsf{n}$ density matrix. The algorithm picks a random diagonal version, then applies a random unitary to it. (R) for real version (F) for fractional (real) version, and optional specifcation of the rank, rank as the final argument.

(R)PickRandomPOVM[dim, num]: Outputs a random num element POVM with dimension dim, i.e., a set of num matrices E_i such that $\sum_i E_i = \mathbb{1}$. [Fractional (F) version not currently implemented.]

(R)PickRandomMeasurement[dim, num]: Outputs a random num element measurement with dimension dim, i.e., a set of matrices K_i such that $\sum_i \mathsf{K}_i^\dagger \mathsf{K}_i = \mathbb{1}$. ($\mathsf{K}_i^\dagger \mathsf{K}_i$ are the POVM elements.) [Fractional (F) version not currently implemented.]

(R/F)PickRandomChannel[dim$_1$, dim$_2$, num (,prec)]: Outputs the Kraus form of a random num element channel from dimension dim$_1$ to dimension dim$_2$, i.e., a set of num matrices K_i such that $\sum_i \mathsf{K}_i^\dagger \mathsf{K}_i = \mathbb{1}$. In other words, the intended channel is $\rho \mapsto \sum_i \mathsf{K}_i \rho \mathsf{K}_i^\dagger$.

C. Distance Measures And Distinguishing States

Dist[ρ_1, ρ_2]: This computes the trace distance between matrices ρ_1 and ρ_2, i.e. $\frac{1}{2}\mathrm{tr}\,|\rho_1 - \rho_2|$.

QuickDist[ρ_1, ρ_2]: This computes the trace distance in a different (usually faster) way. The difference between these two methods is that the second finds the eigenvalues and works on these, while the first relies on Mathematica's built in MatrixPower command.

Fidelity[ρ_1, ρ_2]: This computes the fidelity between the two states, i.e. $\mathrm{tr}\sqrt{\sqrt{\rho_1}\rho_2\sqrt{\rho_1}}$.

OptimumPOVM[ρ_1, ρ_2, p$_1$]: This returns $\{\mathsf{E}_0, \mathsf{E}_1\}$, where these are the POVM elements that optimally distinguish ρ_1 and ρ_2, where ρ_1 occurs with probability p_1, and ρ_2 with probability $1 - \mathsf{p}_1$. One can enter OptimumPOVM[ρ_1, ρ_2], in which case it is assumed that $\mathsf{p}_1 = \frac{1}{2}$. The POVM is optimal in the sense that it maximizes the probability of correctly guessing whether a given state is ρ_1 or ρ_2.

D. Entropy Measures

The entropy commands in this section may not work for symbolic matrices because they use If statements to distinguish cases and the conditions may not be resolvable for symbolic values.

Matrixlog[M]: Returns the natural logarithm of matrix M, where M is Hermitian.

Matrixlog[b, M]: Returns the logarithm of matrix M in base b, where M is Hermitian.

Matrixxlogx[M]: Returns the function $x \log x$ applied to Hermitian matrix M, taking $0 \log 0$ to be 0.

Matrixxlogx[b, M]: Returns the function $x \log_b x$ applied to Hermitian matrix M, taking $0 \log_b 0$ to be 0.

ShanEntropy[problist]: Computes the Shannon entropy of the distribution given by problist $= \{\mathsf{p}_1, \mathsf{p}_2, \ldots\}$, i.e. $\sum_i -p_i \log_2 p_i$.

ShanEntropy[p]: This is equivalent to ShanEntropy[{p, 1-p}].

vNEntropy[rho, keep, desc]: Computes the (conditional) von Neumann entropy of rho, a multi-partite system whose dimensions are given by desc, conditioning on the systems specified by keep which uses 1 for the system, 2 for systems to condition on and 0 for systems to trace out. Example: vNEntropy[rho, {1,0,2}, {2,4,2}] takes a $2 \times 4 \times 2$ dimensional matrix rho, traces out the 4 dimensional subsystem, then calculates the von Neuman entropy of the first system given the third.

vNInfo[rho, keep, desc]: Computes the (conditional) von Neumann mutual information of rho, a multi-partite system whose dimensions are given by desc, conditioning on the systems specified by keep which uses 1 for the first system, 2 for the second system, 3 for systems to condition on and 0 for systems to trace out. Example: vNInfo[rho, {1,2,0,3}, {2,4,2,2}] takes a $2 \times 4 \times 2 \times 2$ dimensional matrix rho and computes the quantum mutual information between the first and second systems given the fourth.

RelEnt[A, B]: Computes the von Neumann relative entropy of A and B, equal to $(\mathrm{tr}(\mathsf{A} \log \mathsf{A}) - \mathrm{tr}(\mathsf{A} \log \mathsf{B}))/\mathrm{tr}(\mathsf{A})$. If A is not contained in the support of B then the answer is infinite.

RelEnt[α, A, B]: Computes the α relative entropy of A and B, equal to $\frac{1}{\alpha-1}\log_2 \mathrm{tr}(\mathsf{A}^\alpha \mathsf{B}^{1-\alpha})$. For $\alpha = 0$ this evaluates the relative entropy and for $\alpha = 0$, the expression is interpreted via the limit $\alpha \to 0^+$.

RenyiEnt[α, rho, keep, desc]: Computes the α Rényi entropy of rho, a multi-partite system whose dimensions are given by desc, conditioning on the systems specified by keep which uses 1 for the system, 2 for systems to condition on and 0 for systems to trace out. Here the conditional Rényi entropy is defined by $H_\alpha(A|B) = \frac{1}{1-\alpha}\log_2 \mathrm{tr}\left(\rho_{AB}^\alpha(\mathbb{1}\otimes\rho_B)^{1-\alpha}\right)$. For $\alpha = 1$ this evaluates the von Neumann entropy, and for $\alpha = 0$, the expression is interpreted via the limit $\alpha \to 0^+$.

E. Linear Programming

(N)RemoveIneqConstraints[M, b]: Removes redundant inequality constraints from the set of equations specified by matrix M where $\mathsf{M}\,\mathsf{x} \geq \mathsf{b}$, returning $\{\mathsf{M}', \mathsf{b}'\}$. The parameter b can also be given in the form of pairs of values $\{\{\mathsf{b}_1, \mathsf{s}_1\}, \{\mathsf{b}_2, \mathsf{s}_2\}\ldots\}$ where $\mathsf{s}_\mathsf{i} \in \{-1, 0, 1\}$ represents less than or equal to, equal to, or greater than (the syntax is the same as in Mathematica's LinearProgramming command). Equality constraints are retained. This works by optimizing each constraint subject to the others to see whether it is already implied. (N) is for the numerical version, where each LinearProgramming minimization is done numerically (which is faster). It may also be desirable to use RemoveDuplicateConstraints[M, b] first to speed up the process.

RemoveDuplicateConstraints[M, b]: Removes redundant constraints from the set of equations specified by matrix M where $\mathsf{M}\,\mathsf{x} \geq \mathsf{b}$, returning $\{\mathsf{M}', \mathsf{b}'\}$. The parameter b can also be given in the form of pairs of values $\{\{\mathsf{b}_1, \mathsf{s}_1\}, \{\mathsf{b}_2, \mathsf{s}_2\}\ldots\}$ where $\mathsf{s}_\mathsf{i} \in \{-1, 0, 1\}$ represents less than or equal to, equal to, or greater than (the syntax is the same as in Mathematica's LinearProgramming command). This works by checking for identical rows in M and processing them accordingly (if $\mathsf{M}\,\mathsf{x} = 5$, any inequality constraints $\mathsf{M}\,\mathsf{x} \geq c$ are removed, if $\mathsf{M}\,\mathsf{x} \geq 4$ and $\mathsf{M}\,\mathsf{x} \geq 5$, then the former is removed). Note that no consistency check is performed on the equations, so an inconsistent set may become consistent, since if $\mathsf{M}\,\mathsf{x} = 3$ and $\mathsf{M}\,\mathsf{x} = 4$, then one of these will be removed.

Prep[M, b]: Here b must have the form $\{\{\mathsf{b}_1, \mathsf{s}_1\}, \{\mathsf{b}_2, \mathsf{s}_2\}\ldots\}$ (the syntax is the same as in Mathematica's LinearProgramming command) and is converted to $\{\mathsf{M}', \mathsf{b}'\}$, where $\mathsf{M}'\,\mathsf{x} \geq \mathsf{b}'$. Thus, if s_1 is negative the sign is flipped and if $\mathsf{s}_1 = 0$, two inequalities are included corresponding to $\mathsf{M}_\mathsf{i}\,\mathsf{x} \geq \mathsf{b}_\mathsf{i}$ and $-\mathsf{M}_\mathsf{i}\,\mathsf{x} \geq -\mathsf{b}_\mathsf{i}$.

FourierMotzkin[M, b, elim, (Options)]: Here b must have the form $\{\mathsf{b}_1, \mathsf{b}_2, \ldots\}$ and signifies $\mathsf{M}\,\mathsf{x} \geq \mathsf{b}$ componentwise. The command produces a new set of inequalities eliminating the variables specified by elim, e.g., if elim={1,4}, then the first and fourth variables are eliminated. The command returns $\{\mathsf{M}', \mathsf{b}'\}$ such that $\mathsf{M}'\,\mathsf{x}' \geq \mathsf{b}'$, where x′ now does not involve the eliminated variables. This command tries to optimize the order of the eliminations (choosing the next variable to keep the total number of inequalities smallest), and uses RemoveIneqConstraints[M, b] after each step to reduce the number before starting the next elimination [to turn this off, use the option Simp→False]. To go through the eliminations one at a time without removing redundant equations, use Elim[M, b, i] to eliminate the ith variable only. Note that the Elim function adds in a row to the matrix corresponding to the positivity of the element to be removed (such a constraint is assumed by LinearProgramming) [to turn this off, use the option AddId→False].

[Commands NegInstances[M, i] and PosInstances[M, i] are private for Elim.]

F. Miscellaneous

IntDigs[num,bases]: Writes the number num in terms of the bases specified in bases. The length of the output is always equal to the number of bases. E.g., IntDigs[16,{4,2,3}] gives $\{2, 1, 1\}$ meaning that $16 = 2\times(2\times3)+1\times3+1\times1$. Note that IntDigs[24,{4,2,3}] gives $\{0, 0, 0\}$ since the most significant digit cannot be larger than 4. The next digit would give the number of 4.3.2=24s.

Progress[i, i_min, i_max, num]: This command is useful to give a progress report in large runs. Assuming i increases by an integer each time, and runs between i_min and i_max, this will display num outputs throughout this range. For example, to get a progress indication every 10%, use num=10. [N.B. This command may also work if i increments by more than an integer each time, provided num is not too large.]

ProgressTemporary[i, i_min, i_max, num]: As above but uses PrintTemporary so that the progress display disappears on completion.

(N)ThreadSolve[eqns, soln, var]: Solves the set of equations eqns[[i]]=soln for variables var. Example: ThreadSolve$[\{a+1/2, 6*a+3/2\}, 0, a]$ returns $\{-1/2, -1/4\}$, these being the values of a that solve the first and second equations. (N) is for the numerical version.

σ[i]: This outputs the i^{th} Pauli Matrix, with $i \in \{0, 1, 2, 3\}$. The σ symbol is in the basic calculations palette, or can be entered using the keyboard sequence ESCAPE, s, ESCAPE.

Support[M]: Returns the projector onto the support of a matrix M.

PosPart[M]: Returns the (strictly) positive part of a matrix M. Do not use with symbolic matrices.

PosPart[M, assum]: Returns the (strictly) positive part of a matrix M, making assumptions assum about certain variables in M. Example: PosPart[{{Sin[t]^2,-Sin[t]*Cos[t]},{-Sin[t]*Cos[t],-Sin[t]^2}}] returns {0,0} with a warning message, but PosPart[{{Sin[t]^2,-Sin[t]*Cos[t]},{-Sin[t]*Cos[t],-Sin[t]^2}},0¡t¡1] returns a meaningful answer. Care is needed for use with symbolic matrices.

NegPart[M]: Returns the (strictly) negative part of a matrix M. Do not use with symbolic matrices.

NegPart[M, assum]: Returns the (strictly) negative part of a matrix M, making assumptions assum about certain variables in M. Care is needed for use with symbolic matrices.

AntiDiagonalMatrix[M]: Turns a matrix into a vector of diagonal values. Example: AntiDiagonalMatrix$[\{\{a, b\}, \{c, d\}\}]$ returns $\{\{a\}, \{d\}\}$. Note: Mathematica's Diagonal command is similar, up to the form of the output.

CreateMatrix[W, r, c]: This creates the $r \times c$ matrix with elements W[i, j].

CreateHermitianMatrix[W, r, c]: This acts like CreateMatrix except that it ensures the output is Hermitian. May be appended to CreateHermitianMatrixR[W, r, c] to further ensure that the variables used are real, i.e. the elements are now WR[i, j]+i WC[i, j] (and WC[i, i]=0).

CreateSymmetricMatrix[W, r, c]: As CreateHermitianMatrix[W, r, c] but the matrix created is symmetric rather than Hermitian.

RemoveRepeatedInstances[list]: This returns a smaller list which features only one instance of each entry. The order is preserved based on the order in which the first instance occurred in the original list. Example: RemoveRepeatedInstances$[\{a, b, b, a, 1, s, 1\}]$ returns $\{a, b, 1, s\}$. Note: Mathematica's DeleteDuplicates command also does this.

SetMinus[S1, S2]: If S1 and S2 are two lists, this computes S1\S2, i.e. it returns S1 with the elements of S2 removed. Note that if S1 is not a set, e.g., it contains repeated entries, then so does the output. The order is maintained. Example: SetMinus[{1,2,3,4},{2,3}] returns {1,4} .

MajorizeQ[A, B]: Checks whether A majorizes B, where A and B are lists of numbers, i.e. whether when ordered by size with $a_1 \geq a_2 \geq \ldots$, and $b_1 \geq b_2 \geq \ldots$, we have $a_1 \geq b_1$, $a_1 + a_2 \geq b_1 + b_2$ etc.

ManualDeriv[fn, vars, point, steps]: Computes an estimate of the derivative of the function fn, which should be a function of the variables in set vars at the point point with set of stepsizes steps. This is useful for complicated expressions that cannot be differentiated symbolically. The function is best input within Hold[ ]. E.g., ManualDeriv[Hold[Integrate[y*x, {x, 0, a}, {y, b, c}]], {a, b, c}, {3, 2, 3}, {1/100, 1/10, 1/10}] treats the integral as a function of a, b, c and uses steps 1/100, 1/10, 1/10. The first part of the output is (f[3+1/100, 2, 3] - f[3, 2, 3]) / (1/100) etc., where f[a,b,c] is the integral.

QRDecomp[M]: For a square matrix M, returns square matrices {Q, R} such that Q is unitary, R is right (upper) triangular and Q†R=M. Note: Mathematica's built in QRDecomposition does not output square matrices if M doesn't have full rank.

RQDecomp[M]: For a square matrix M, returns square matrices {R, Q} such that R is right (upper) triangular, Q is unitary and $RQ^\dagger$=M.

QLDecomp[M]: For a square matrix M, returns square matrices {Q, L} such that Q is unitary, L is left (lower) triangular and $Q^\dagger L$=M.

LQDecomp[M]: For a square matrix M, returns square matrices {L, Q} such that L is left (lower) triangular, Q is unitary and $LQ^\dagger$=M.

QRDecompPos[M]: As QRDecomp but ensures that R has positive diagonal entries. [Similarly, RQDecompPos, LQDecompPos and QLDecompPos.]

[These use the private function SignZeroPos[v] which takes a list v and outputs a list of the signs of (the real parts of) v but taking 0 to have positive sign, e.g. SignZeroPos[{1,0,-1,-2+I,0}] gives {1,-1,-1,-1,-1}.]

QRDecompNeg[M]: As QRDecomp but ensures that R has negative diagonal entries. [Similarly, RQDecompNeg, LQDecompNeg and QLDecompNeg.]

[These use the private function SignZeroNeg[v] which takes a list v and outputs a list of the signs of (the real parts of) v but taking 0 to have negative sign, e.g. SignZeroPos[{1,0,-1,-2+I,0}] gives {1,-1,-1,-1,-1}.]