QuantumDDInflation_symmetrized

April 14, 2021

[27]:
```
run ~/w/replab/replab_init.m
addpath(genpath('~/software/QETLAB'));
replab.globals.useReconstruction(1);
replab.globals.verbosity(0);
profile on
```

1 Symmetrization of the quantum inflation SDP for the GHZ state

We sketch below the construction of a symmetry-adapted SDP for that optimization. The resulting SDP has blocks of maximal size `5 x 5`. We describe all the spaces present, and their symmetries.

1.1 Spaces of Hermitian matrices

- Triangle: Three qubits `A B C`, the state we are testing which has the symmetries of the GHZ state, see `GHZ_symmetries.ipynb`
- Two triangles: Six qubits `A1 B1 C1 A2 B2 C2`
- Two triangles after partial transposition of `A2 B2 C2`: Six qubits
- Ring: Six qubits, in order `A1 B1 C2 A2 B2 C2`
- `ACAC` space: Four qubits `A1 C1 A2 C2`, obtained by tracing out `B1 B2` in the ring. Matches the `A1 C2 A2 C1` qubits in the two triangles space.
- `ABCB` space: Four qubits `A1 B1 C1 B2`, after tracing out `A2 C2` in the ring
- `ABCBT` space: Four qubits `A1 B1 C1 B2`, obtained from `ABCB` after partial transpose of `B2`

1.2 Symmetry groups

For the symmetries, we see that we have the symmetries of the GHZ state, and in addition we have a shift group $H = S(2)$ that acts by either permuting the two triangles, or cycling the subsystems in the ring by three positions. To recap:

- P is the symmetry group $S(3)$ permuting the subsystems `A B C`, with P_{AC} the subgroup permuting only `A C`. We generate P using the three transpositions `(A,B)`, `(A,C)`, `(B,C)`.
- H the shift group $S(2)$ shifting the ring by $+3$, or permuting the two triangles
- L the symmetry group $S(2)$ permuting the levels
- D is the direct product $P \times H \times L$

- D_{AC} is the direct product $P_{AC} \times H \times L$
- T the torus group acting on the phases of rank $r = 4$ and dimension $n = 6$.
- G is the semidirect product $D \ltimes T$, the most general symmetry group we employ here.
- $G_{AC} = D_{AC} \ltimes T$ is a subgroup of G employed when the spaces break the symmetry of permuting subsystems. We also construct the injection from G_{AC} to G, handy when mix'n'matching symmetric spaces.

1.3 Spaces with symmetries under G

- Triangle: The space is invariant under G, but H has no effect on the three subsystems.
- Two triangles: P permutes the subsystems inside each triangle, while H swaps the two triangles.
- Two triangles after partial transposition of `A2 B2 C2`: same. Note that the states can all be considered real wlog, the partially transposed state can still be assumed symmetric under the swap of triangles. Alternatively, we could write a space where H does not act (but we already have so many spaces here!).
- Ring: We generate P using the reflections that swap pairs of parties. For example, the transposition `(A,C)` is realized by permuting $A_1 \leftrightarrow C_2$, $C_1 \leftrightarrow A_2$, and it also swaps $B_1 \leftrightarrow B_2$. The same for the other two transpositions, and one realizes that the permutation group on six elements thus generated is isomorphic to S_3. Then H is simplify the shift `A1 B1 C1 A2 B2 C2` to `A2 B2 C2 A1 B1 C1`.

1.4 Spaces with symmetries under G_{AC}

For all the spaces below, the only permutation of subsystems allowed is the transposition of `A` and `C`. For that reason, those spaces will be defined using the group G_{AC}.

- `ACAC` space: T, H and L still act as before; the action of P_{AC} is modeled around its action on the ring.
- `ABCB` space: T, L still act as before; the action of P_{AC} permutes the two `A`, `C` subsystems, while H is present but does not act.
- `ABCBT` space: same as `ABCB` except that the torus group has a complex conjugate action on the transposed subsystem.

2 Definitions

Now let us write all of that.

2.1 Groups

[28]:
```
T6 = replab.T(6);
T6 = T6.withNames({'a0' 'b0' 'c0' 'a1' 'b1' 'c1'});
T = T6.subgroupWith('a0*b0*c0 = 1', 'a1*b1*c1 = 1');
P = replab.S(3);
```

```
P_AC = P.subgroup({[3 2 1]});
L = replab.S(2);
H = replab.S(2);
D = P.directProduct(H, L);
D_AC = P_AC.directProduct(H, L);
D_AC_inj = D_AC.morphismByFunction(D, @(x) x); % injection from D_AC to D;␣
 ↪elements have the same shape
```

2.1.1 Semidirect product constructions

We start with the torus automorphisms, which correspond to generators of D.

[29]:
```
% Permutation of AB
gAB = {[2 1 3] [1 2] [1 2]};
actAB = T.automorphism('b0', 'a0', 'c0', 'b1', 'a1', 'c1');
% Permutation of AC
gAC = {[3 2 1] [1 2] [1 2]};
actAC = T.automorphism('c0', 'b0', 'a0', 'c1', 'b1', 'a1');
% Permutation of BC
gBC = {[1 3 2] [1 2] [1 2]};
actBC = T.automorphism('a0', 'c0', 'b0', 'a1', 'c1', 'b1');
% Shift
gH = {[1 2 3] [2 1] [1 2]};
actH = T.automorphism('a0', 'b0', 'c0', 'a1', 'b1', 'c1'); % identity
% Permutation of the two levels
gL = {[1 2 3] [1 2] [2 1]};
actL = T.automorphism('a1', 'b1', 'c1', 'a0', 'b0', 'c0');
```

Now, we construct G using a semidirect product (torus groups have a special method to achieve that). We also construct G_{AC} and the injection from G_{AC} to G.

[30]:
```
G = T.semidirectProductByFiniteGroup(D, 'preimages', {gAB, gAC, gBC, gH, gL},␣
 ↪'images', {actAB, actAC, actBC, actH, actL});
G_AC = T.semidirectProductByFiniteGroup(D_AC, 'preimages', {gAC, gH, gL},␣
 ↪'images', {actAC, actH, actL});
G_AC_inj = G_AC.morphismByFunction(G, @(x) x); % we keep the same element
```

2.2 Spaces and objects that live on them

The spaces below are equivariant spaces: they contain Hermitian matrices that commute with a representation of G or G_{AC}, i.e. $[X, \rho_g] = 0$ with $X = X^\dagger$. To construct those spaces, we construct the associated representation ρ, and then use the method `.hermitianInvariant`; it is equivalent to a `.commutant` space when the representation is unitary, but adds the constraint that matrices are Hermitian, which enables the use of SDP constraints later on.

2.2.1 Triangle equivariant space

We start with the torus representation.

[31]:
```
Trep_triangle = T.diagonalRepWith('a0 b0 c0', ...
                                  'a0 b0 c1', ...
                                  'a0 b1 c0', ...
                                  'a0 b1 c1', ...
                                  'a1 b0 c0', ...
                                  'a1 b0 c1', ...
                                  'a1 b1 c0', ...
                                  'a1 b1 c1');
```

And now the representation of the finite group D; note that D always act by permuting the coefficients of the state.

[32]:
```
imgAB_triangle = replab.Permutation.toMatrix([1 2 5 6 3 4 7 8]);
imgAC_triangle = replab.Permutation.toMatrix([1 5 3 7 2 6 4 8]);
imgBC_triangle = replab.Permutation.toMatrix([1 3 2 4 5 7 6 8]);
imgL_triangle = replab.Permutation.toMatrix([8 7 6 5 4 3 2 1]);
imgH_triangle = replab.Permutation.toMatrix([1 2 3 4 5 6 7 8]);
Drep_triangle = D.repByImages('C', 8, 'preimages', {gAB, gAC, gBC, gH, gL},␣
 ↪'images', {imgAB_triangle, imgAC_triangle, imgBC_triangle, imgH_triangle,␣
 ↪imgL_triangle});
```

We assemble the representation of G by combining representations of D and T. Note that the representations involved have to be compatible, and RepLAB does not necessarily check these compatibility conditions.

[33]:
```
rep_triangle = G.semidirectProductRep(Drep_triangle, Trep_triangle);
```

Now the space of Hermitian matrices that commute with `rep_triangle`:

[34]:
```
E_triangle = rep_triangle.hermitianInvariant;
```

It's time to define a few objects that live on the space `E_triangle`.

We define the GHZ state and the maximally mixed state. Both commute with `rep_triangle`; thus we can wrap them in a `replab.equivar`. A `replab.equivar` is like a `sdpvar`, but it keeps tracks of the equivariant space the variable lives in, and only stores information about the degrees of freedom that remain after imposing the symmetry.

We write the SDP variable `X_triangle` that represents state whose compatibility we are testing.

[35]:
```
ghz = [1 0 0 0 0 0 0 1]'*[1 0 0 0 0 0 0 1]/2;
ghz = replab.equivar(E_triangle, 'value', ghz);
noise = eye(8)/8;
noise = replab.equivar(E_triangle, 'value', noise);
lambda = sdpvar;
X_triangle = ghz * (1-lambda) + noise * lambda; % note that we multiply with␣
 ↪the sdpvar on the right; the reverse would not work due to class precedence␣
 ↪issues
```

… and now we write the beginning of our optimization problem. If we wanted to put a (unnecessary) constraint on the trace in `C`, we would write `trace(sdpvar(X_triangle)) == 1`, where `sdpvar(...)` recover the sdpvar present in a `replab.equivar` so that standard MATLAB/YALMIP operations can be used. Note that we do not write `sdpvar(X_triangle) >= 0` but rather `issdp(X_triangle)`; the later is a very efficient way of writing the semidefinite positiveness constraint by using the block-diagonal structure of `X_triangle`. Note that here, instead of a `8 x 8` SDP constraint, we get three linear inequalities – corresponding to three `1 x 1` blocks.

[36]:
```
obj = lambda;
C = [issdp(X_triangle)]
```

```
+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
|   ID|                    Constraint|     Coefficient range|
+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
|   #1|   Element-wise inequality 1x1|            0.875 to 1|
|   #2|   Element-wise inequality 1x1|   2.4652e-32 to 0.125|
|   #3|   Element-wise inequality 1x1|        0.125 to 0.125|
+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

2.2.2 Two triangles equivariant space

Onwards to our next space! It contains six qubits: the first three `A1 B1 C1` correspond to one copy of the state, the next three `A2 B2 C2` to the second copy. We have no way to impose that the density matrix in that space factorizes across the two triangles; but we will impose a PT constraint.

Apart from the action of H, the representations here are simply a tensor product of two copies of the representations acting on a single triangle.

[37]:
```
Trep_twotriangles = kron(Trep_triangle, Trep_triangle);
imgAB_twotriangles = kron(imgAB_triangle, imgAB_triangle);
imgAC_twotriangles = kron(imgAC_triangle, imgAC_triangle);
imgBC_twotriangles = kron(imgBC_triangle, imgBC_triangle);
imgH_twotriangles = reshape(permute(reshape(eye(64), [8 8 64]), [2 1 3]), [64␣
 ↪64]);
imgL_twotriangles = kron(imgL_triangle, imgL_triangle);
Drep_twotriangles = D.repByImages('C', 64, 'preimages', {gAB, gAC, gBC, gH,␣
 ↪gL}, 'images', {imgAB_twotriangles, imgAC_twotriangles, imgBC_twotriangles,␣
 ↪imgH_twotriangles, imgL_twotriangles});
rep_twotriangles = G.semidirectProductRep(Drep_twotriangles, Trep_twotriangles);
E_twotriangles = rep_twotriangles.hermitianInvariant;
```

While each of the two triangles, seen individually, match exactly `X_triangle`, there are additional degrees of freedom. We define a `replab.equivar` to parameterize it, and ask it to be SDP.

[38]:
```
X_twotriangles = replab.equivar(E_twotriangles);
C = [C; issdp(X_twotriangles)];
```

Now, we define the partial trace operation that keeps only one copy of the triangle. This map, say F respects the symmetry of the spaces. Let us write $\Psi = F[\Xi]$, with Ξ corresponding to

`X_twotriangles` and Ψ to `X_triangle`. Let ρ be the representation that acts on $\Xi \to \rho_g \Xi \rho_g^\dagger$, and σ the representation that acts on $\Psi \to \sigma_g \Psi \sigma_g^\dagger$.

The map F is equivariant as we observe that $F[\rho_g \Xi \rho_g^\dagger] = \sigma_g F[\Xi] \sigma_g^\dagger$. Thus, we define that partial trace operation as a `replab.equiop`, which is an operator that can be applied on `replab.equivar`s.

Note: when using `equiop` and `equivar`, RepLAB will verify that the groups, equivariant spaces, etc, involved match. It is important to use the *same MATLAB instance* of the mathematical object in all definitions. For some groups/representations, RepLAB is able to check that two instances of the same mathematical object; in other cases not, and RepLAB will check that the internal IDs of objects match.

[39]:
```
op_mapsTriangle = replab.equiop.generic(E_twotriangles, E_triangle, @(X)␣
 ↪PartialTrace(X, [4 5 6], [2 2 2 2 2 2]));
% op_mapsTriangle.check % <- run this if you want to verify the equivariance␣
 ↪property using random tests
C = [C; op_mapsTriangle(X_twotriangles) == X_triangle]
```

```
+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
|    ID|                         Constraint|      Coefficient range|
+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
|    #1|       Element-wise inequality 1x1|             0.875 to 1|
|    #2|       Element-wise inequality 1x1|    2.4652e-32 to 0.125|
|    #3|       Element-wise inequality 1x1|         0.125 to 0.125|
|    #4|   Matrix inequality (complex) 3x3|                 1 to 1|
|    #5|       Element-wise inequality 1x1|                 1 to 1|
|    #6|   Matrix inequality (complex) 2x2|                 1 to 1|
|    #7|       Element-wise inequality 1x1|                 1 to 1|
|    #8|       Element-wise inequality 1x1|                 1 to 1|
|    #9|       Element-wise inequality 1x1|                 1 to 1|
|   #10|       Element-wise inequality 1x1|                 1 to 1|
|   #11|       Element-wise inequality 1x1|                 1 to 1|
|   #12|       Element-wise inequality 1x1|                 1 to 1|
|   #13|   Matrix inequality (complex) 2x2|                 1 to 1|
|   #14|   Matrix inequality (complex) 3x3|                 1 to 1|
|   #15|            Equality constraint 1x1|          0.5 to 2.8957|
|   #16|            Equality constraint 1x1|   2.4652e-32 to 2.8957|
|   #17|            Equality constraint 1x1|             0.125 to 1|
+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

2.2.3 Two triangles, PPT equivariant space

Same story here, except that after partial transposition, the action of the torus second triangle needs a complex conjugation (i.e. to use the dual representation).

[40]:
```
```

```
Trep_ppt = kron(Trep_triangle, dual(Trep_triangle)); % the second copy has a␣
 ↪partial transpose
Drep_ppt = D.repByImages('C', 64, 'preimages', {gAB, gAC, gBC, gH, gL},␣
 ↪'images', {imgAB_twotriangles, imgAC_twotriangles, imgBC_twotriangles,␣
 ↪eye(64), imgL_twotriangles});
rep_ppt = G.semidirectProductRep(Drep_ppt, Trep_ppt);
E_ppt = rep_ppt.hermitianInvariant;
```

Now we write the PPT condition to approximate separability across the two triangles.

[41]:
```
op_ppt = replab.equiop.generic(E_twotriangles, E_ppt, @(X) PartialTranspose(X,␣
 ↪[4 5 6], [2 2 2 2 2 2]), 'supportsSparse', true);
% op_ppt.check % <- run this if you want to verify the equivariance property␣
 ↪using random tests
C = [C; issdp(op_ppt(X_twotriangles))]
```

```
+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
|    ID|                          Constraint|      Coefficient range|
+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
|    #1|       Element-wise inequality 1x1|             0.875 to 1|
|    #2|       Element-wise inequality 1x1|    2.4652e-32 to 0.125|
|    #3|       Element-wise inequality 1x1|         0.125 to 0.125|
|    #4|   Matrix inequality (complex) 3x3|                 1 to 1|
|    #5|       Element-wise inequality 1x1|                 1 to 1|
|    #6|   Matrix inequality (complex) 2x2|                 1 to 1|
|    #7|       Element-wise inequality 1x1|                 1 to 1|
|    #8|       Element-wise inequality 1x1|                 1 to 1|
|    #9|       Element-wise inequality 1x1|                 1 to 1|
|   #10|       Element-wise inequality 1x1|                 1 to 1|
|   #11|       Element-wise inequality 1x1|                 1 to 1|
|   #12|       Element-wise inequality 1x1|                 1 to 1|
|   #13|   Matrix inequality (complex) 2x2|                 1 to 1|
|   #14|   Matrix inequality (complex) 3x3|                 1 to 1|
|   #15|            Equality constraint 1x1|          0.5 to 2.8957|
|   #16|            Equality constraint 1x1|   2.4652e-32 to 2.8957|
|   #17|            Equality constraint 1x1|            0.125 to 1|
|   #18|   Matrix inequality (complex) 3x3|   2.4652e-32 to 1.6718|
|   #19|   Matrix inequality (complex) 3x3|   1.2326e-32 to 1.6369|
|   #20|       Element-wise inequality 1x1|          0.16667 to 1|
|   #21|       Element-wise inequality 1x1|          0.16667 to 1|
|   #22|       Element-wise inequality 1x1|         0.16667 to 0.5|
|   #23|       Element-wise inequality 1x1|         0.16667 to 0.5|
|   #24|       Element-wise inequality 1x1|         0.16667 to 0.5|
|   #25|       Element-wise inequality 1x1|     0.16667 to 0.33333|
|   #26|   Matrix inequality (complex) 5x5|      0.0152 to 0.68979|
+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

2.2.4 Ring equivariant space

Now we move to the ring. The ring contains six qubits labeled `A1 B1 C1 A2 B2 C2`. As mentioned above the action of the permutation of subsystems group P needs to be reconstructed from transpositions; then the group H cyclically shifts the subsystems around the ring.

[42]:
```
Trep_ring = kron(Trep_triangle, Trep_triangle);
img_ring = @(p) reshape(permute(reshape(eye(64), [2 2 2 2 2 2 64]), [fliplr(7 -␣
 ↪p) 7]), [64 64]);
imgAB_ring = img_ring([2 1 6 5 4 3]);
imgAC_ring = img_ring([6 5 4 3 2 1]);
imgBC_ring = img_ring([4 3 2 1 6 5]);
imgH_ring = img_ring([4 5 6 1 2 3]);
imgL_ring = kron(imgL_triangle, imgL_triangle);
Drep_ring = D.repByImages('C', 64, 'preimages', {gAB, gAC, gBC, gH, gL},␣
 ↪'images', {imgAB_ring, imgAC_ring, imgBC_ring, imgH_ring, imgL_ring});
rep_ring = G.semidirectProductRep(Drep_ring, Trep_ring);
E_ring = rep_ring.hermitianInvariant;
```

The ring contains a few unobserved degrees of freedom, so we parameterize it by a SDP matrix.

[43]:
```
X_ring = replab.equivar(E_ring);
C = [C; issdp(X_ring)];
```

2.2.5 ACAC space

The `ACAC` corresponds to the subsystems `A1 C1 A2 C2` of the ring (in that order), and `A1 C2 A2 C1` in the two triangles (in that order).

We have implemented our own function `img_ACAC` to construct matrices that permute subsystems; one could use `PermuteSubsystems` from QETLAB as well.

[44]:
```
Trep_ACAC = T.diagonalRepWith('a0 c0 a0 c0', ...
                              'a0 c0 a0 c1', ...
                              'a0 c0 a1 c0', ...
                              'a0 c0 a1 c1', ...
                              'a0 c1 a0 c0', ...
                              'a0 c1 a0 c1', ...
                              'a0 c1 a1 c0', ...
                              'a0 c1 a1 c1', ...
                              'a1 c0 a0 c0', ...
                              'a1 c0 a0 c1', ...
                              'a1 c0 a1 c0', ...
                              'a1 c0 a1 c1', ...
                              'a1 c1 a0 c0', ...
                              'a1 c1 a0 c1', ...
                              'a1 c1 a1 c0', ...
                              'a1 c1 a1 c1');
```

```
img_ACAC = @(p) reshape(permute(reshape(eye(16), [2 2 2 2 16]), [fliplr(5 - p)␣
↪5]), [16 16]);
imgAC_ACAC = img_ACAC([4 3 2 1]);
imgH_ACAC = img_ACAC([3 4 1 2]);
imgL_ACAC = kron(kron([0 1; 1 0], [0 1; 1 0]), kron([0 1; 1 0], [0 1; 1 0]));
Drep_ACAC = D_AC.repByImages('C', 16, 'preimages', {gAC, gH, gL}, 'images',␣
↪{imgAC_ACAC, imgH_ACAC, imgL_ACAC});
rep_ACAC = G_AC.semidirectProductRep(Drep_ACAC, Trep_ACAC);
E_ACAC = rep_ACAC.hermitianInvariant;
```

Now the tricky part. We define the map that singles out the subsystems in the ring, and in the two triangles, to finally match the result as an equality constraint.

Notice that `E_ACAC` is invariant under G_{AC}, while `E_ring` and `E_twotriangles` are invariant under G. We can define a `replab.equiop` that break a symmetry; in that case, we need to specify the relation between the group and its subgroup using an injection from the subgroup to the group – which is the additional argument `sourceInjection` passed to `replab.equiop.generic(...)`.

Here are our two partial trace maps.

[45]:
```
op_ring_to_ACAC = replab.equiop.generic(E_ring, E_ACAC, @(X) PartialTrace(X, [2␣
↪5], [2 2 2 2 2 2]), 'sourceInjection', G_AC_inj);
op_twotriangles_to_ACAC = replab.equiop.generic(E_twotriangles, E_ACAC, @(X)␣
↪PartialTrace(PermuteSystems(X, [1 6 4 3 2 5], [2 2 2 2 2 2]), [5 6], [2 2 2␣
↪2 2 2]), 'sourceInjection', G_AC_inj);
```

Constraint: the marginals match.

[46]:
```
C = [C; op_ring_to_ACAC(X_ring) == op_twotriangles_to_ACAC(X_twotriangles)]
```

```
+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
|    ID|                           Constraint|      Coefficient range|
+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
|    #1|         Element-wise inequality 1x1|             0.875 to 1|
|    #2|         Element-wise inequality 1x1|    2.4652e-32 to 0.125|
|    #3|         Element-wise inequality 1x1|         0.125 to 0.125|
|    #4|     Matrix inequality (complex) 3x3|                 1 to 1|
|    #5|         Element-wise inequality 1x1|                 1 to 1|
|    #6|     Matrix inequality (complex) 2x2|                 1 to 1|
|    #7|         Element-wise inequality 1x1|                 1 to 1|
|    #8|         Element-wise inequality 1x1|                 1 to 1|
|    #9|         Element-wise inequality 1x1|                 1 to 1|
|   #10|         Element-wise inequality 1x1|                 1 to 1|
|   #11|         Element-wise inequality 1x1|                 1 to 1|
|   #12|         Element-wise inequality 1x1|                 1 to 1|
|   #13|     Matrix inequality (complex) 2x2|                 1 to 1|
|   #14|     Matrix inequality (complex) 3x3|                 1 to 1|
|   #15|              Equality constraint 1x1|          0.5 to 2.8957|
|   #16|              Equality constraint 1x1|   2.4652e-32 to 2.8957|
```

```
|   #17|             Equality constraint 1x1|             0.125 to 1|
|   #18|     Matrix inequality (complex) 3x3|   2.4652e-32 to 1.6718|
|   #19|     Matrix inequality (complex) 3x3|   1.2326e-32 to 1.6369|
|   #20|         Element-wise inequality 1x1|          0.16667 to 1|
|   #21|         Element-wise inequality 1x1|          0.16667 to 1|
|   #22|         Element-wise inequality 1x1|        0.16667 to 0.5|
|   #23|         Element-wise inequality 1x1|        0.16667 to 0.5|
|   #24|         Element-wise inequality 1x1|        0.16667 to 0.5|
|   #25|         Element-wise inequality 1x1|    0.16667 to 0.33333|
|   #26|     Matrix inequality (complex) 5x5|     0.0152 to 0.68979|
|   #27|     Matrix inequality (complex) 3x3|                1 to 1|
|   #28|         Element-wise inequality 1x1|                1 to 1|
|   #29|     Matrix inequality (complex) 2x2|                1 to 1|
|   #30|         Element-wise inequality 1x1|                1 to 1|
|   #31|         Element-wise inequality 1x1|                1 to 1|
|   #32|         Element-wise inequality 1x1|                1 to 1|
|   #33|         Element-wise inequality 1x1|                1 to 1|
|   #34|         Element-wise inequality 1x1|                1 to 1|
|   #35|         Element-wise inequality 1x1|                1 to 1|
|   #36|     Matrix inequality (complex) 2x2|                1 to 1|
|   #37|     Matrix inequality (complex) 3x3|                1 to 1|
|   #38|             Equality constraint 1x1|       4.5375e-17 to 2|
|   #39|             Equality constraint 1x1|       2.5673e-16 to 2|
|   #40|   Equality constraint (complex) 2x2|       4.5375e-17 to 2|
|   #41|             Equality constraint 1x1|              0.5 to 1|
|   #42|             Equality constraint 1x1|                1 to 2|
|   #43|             Equality constraint 1x1|          0.10053 to 1|
|   #44|             Equality constraint 1x1|          0.10053 to 1|
+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

2.2.6 ABCB space

This space corresponds to the subsystems `A1 B1 C1 B2` in the ring, in that order.

[47]:
```
Trep_ABCB = T.diagonalRepWith('a0 b0 c0 b0', ...
                              'a0 b0 c0 b1', ...
                              'a0 b0 c1 b0', ...
                              'a0 b0 c1 b1', ...
                              'a0 b1 c0 b0', ...
                              'a0 b1 c0 b1', ...
                              'a0 b1 c1 b0', ...
                              'a0 b1 c1 b1', ...
                              'a1 b0 c0 b0', ...
                              'a1 b0 c0 b1', ...
                              'a1 b0 c1 b0', ...
                              'a1 b0 c1 b1', ...
                              'a1 b1 c0 b0', ...
                              'a1 b1 c0 b1', ...
```

```
                                'a1 b1 c1 b0', ...
                                'a1 b1 c1 b1');
img_ABCB = @(p) reshape(permute(reshape(eye(16), [2 2 2 2 16]), [fliplr(5 - p)␣
↪5]), [16 16]);
imgAC_ABCB = img_ABCB([3 2 1 4]);
imgH_ABCB = eye(16);
imgL_ABCB = kron(kron([0 1; 1 0], [0 1; 1 0]), kron([0 1; 1 0], [0 1; 1 0]));
Drep_ABCB = D_AC.repByImages('C', 16, 'preimages', {gAC, gH, gL}, 'images',␣
↪{imgAC_ABCB, imgH_ABCB, imgL_ABCB});
rep_ABCB = G_AC.semidirectProductRep(Drep_ABCB, Trep_ABCB);
E_ABCB = rep_ABCB.hermitianInvariant;
```

We define the (equivariant) partial trace map. It is useless on its own, but we will combine it with the partial transpose of the unconnected `B` subsystem.

[48]:
```
op_ABCB = replab.equiop.generic(E_ring, E_ABCB, @(X) PartialTrace(X, [4 6], [2␣
↪2 2 2 2 2]), 'sourceInjection', G_AC_inj);
```

2.2.7 ABCB^T space

Corresponds to A1B1C1B2 in the ring, with the B2 system partial transposed.

[49]:
```
Trep_ABCBT = T.diagonalRepWith('a0 b0 c0 b0^-1', ...
                               'a0 b0 c0 b1^-1', ...
                               'a0 b0 c1 b0^-1', ...
                               'a0 b0 c1 b1^-1', ...
                               'a0 b1 c0 b0^-1', ...
                               'a0 b1 c0 b1^-1', ...
                               'a0 b1 c1 b0^-1', ...
                               'a0 b1 c1 b1^-1', ...
                               'a1 b0 c0 b0^-1', ...
                               'a1 b0 c0 b1^-1', ...
                               'a1 b0 c1 b0^-1', ...
                               'a1 b0 c1 b1^-1', ...
                               'a1 b1 c0 b0^-1', ...
                               'a1 b1 c0 b1^-1', ...
                               'a1 b1 c1 b0^-1', ...
                               'a1 b1 c1 b1^-1');
rep_ABCBT = G_AC.semidirectProductRep(Drep_ABCB, Trep_ABCBT);
E_ABCBT = rep_ABCBT.hermitianInvariant;
```

Now we define the PPT constraint.

[50]:
```
op_ABCBT = replab.equiop.generic(E_ABCB, E_ABCBT, @(X) PartialTranspose(X, [4],␣
↪[2 2 2 2]));
C = [C; issdp(op_ABCBT(op_ABCB(X_ring)))];
```

3 Running the optimization problem

[51]:
```
optimize(C, lambda)
```

```
 num. of constraints = 67
 dim. of sdp    var  = 68,   num. of sdp  blk  = 12
 dim. of linear var  = 27
 dim. of free   var  = 17
 *** convert ublk to linear blk
******************************************************************************
************
   SDPT3: homogeneous self-dual path-following algorithms
******************************************************************************
************
 version  predcorr  gam  expon
   HKM      1      0.000   1
it pstep dstep pinfeas dinfeas  gap     mean(obj)    cputime    kap   tau
theta
------------------------------------------------------------------------------
------------
 0|0.000|0.000|1.4e+02|4.9e+01|1.1e+04| 4.949494e+00|
0:0:00|1.1e+04|1.0e+00|1.0e+00| chol 1  1
 1|0.031|0.031|1.4e+02|4.9e+01|1.2e+04| 5.261026e+00|
0:0:00|1.1e+04|1.0e+00|1.0e+00| chol 1  1
 2|0.088|0.088|1.4e+02|4.8e+01|1.2e+04| 6.135195e+00|
0:0:00|1.1e+04|1.0e+00|9.9e-01| chol 1  1
 3|0.272|0.272|1.3e+02|4.4e+01|1.3e+04| 8.191503e+00|
0:0:00|1.0e+04|1.0e+00|9.1e-01| chol 1  1
 4|0.757|0.757|7.7e+01|2.7e+01|1.1e+04| 1.789587e+01|
0:0:00|5.4e+03|7.4e-01|4.1e-01| chol 1  1
 5|1.000|1.000|1.9e+01|6.7e+00|3.6e+03| 2.346262e+01|
0:0:00|1.0e+03|6.2e-01|8.5e-02| chol 1  1
 6|0.916|0.916|1.6e+00|5.8e-01|3.1e+02| 2.014538e+01|
0:0:00|1.8e+01|6.7e-01|7.9e-03| chol 1  1
 7|0.640|0.640|6.1e-01|2.2e-01|8.1e+01| 9.373543e+00|
0:0:00|7.4e-01|9.3e-01|4.1e-03| chol 1  1
 8|1.000|1.000|6.4e-02|2.3e-02|6.7e+00| 1.129130e+00|
0:0:00|1.3e-01|1.5e+00|6.9e-04| chol 1  1
 9|0.858|0.858|1.5e-02|6.3e-03|1.3e+00|-3.349431e-01|
0:0:00|5.7e-02|1.8e+00|2.0e-04| chol 1  1
10|0.823|0.823|1.0e-02|4.7e-03|9.0e-01|-1.436469e-01|
0:0:00|2.5e-02|1.8e+00|1.3e-04| chol 1  1
11|0.936|0.936|1.5e-03|2.4e-03|1.2e-01|-3.560730e-01|
0:0:00|1.2e-02|1.9e+00|2.0e-05| chol 1  1
12|1.000|1.000|3.2e-04|2.1e-03|2.8e-02|-3.635451e-01|
0:0:00|1.8e-03|1.9e+00|4.4e-06| chol 1  1
```

```
13|0.871|0.871|4.9e-05|1.9e-03|4.3e-03|-3.633393e-01|
0:0:00|6.0e-04|1.9e+00|6.8e-07| chol 1  1
14|1.000|1.000|2.7e-05|7.3e-04|2.3e-03|-3.662572e-01|
0:0:00|6.8e-05|1.9e+00|3.7e-07| chol 1  1
15|0.874|0.874|4.7e-06|3.5e-04|4.0e-04|-3.670573e-01|
0:0:00|3.9e-05|1.9e+00|6.5e-08| chol 1  1
16|1.000|1.000|6.9e-07|1.2e-04|6.0e-05|-3.676049e-01|
0:0:00|6.1e-06|1.9e+00|9.6e-09| chol 1  1
17|1.000|1.000|2.2e-07|4.6e-05|1.9e-05|-3.677728e-01|
0:0:00|9.3e-07|1.9e+00|3.1e-09| chol 1  1
18|1.000|1.000|5.9e-08|1.8e-05|5.1e-06|-3.678396e-01|
0:0:00|2.9e-07|1.9e+00|8.2e-10| chol 1  1
19|1.000|1.000|1.5e-08|7.4e-06|1.3e-06|-3.678664e-01|
0:0:00|7.7e-08|1.9e+00|2.2e-10| chol 1  1
20|1.000|1.000|3.4e-09|3.0e-06|3.0e-07|-3.678772e-01|
0:0:00|2.0e-08|1.9e+00|4.8e-11|# chol 1  1
21|1.000|1.000|1.2e-10|5.9e-08|8.5e-09|-3.678842e-01|
0:0:00|4.6e-09|1.9e+00|1.6e-12|
  Stop: max(relative gap,infeasibilities) < 1.00e-07
------------------------------------------------------------------
 number of iterations   = 21
 primal objective value = -3.67884309e-01
 dual   objective value = -3.67884029e-01
 gap := trace(XZ)       = 8.46e-09
 relative gap           = 6.18e-09
 actual relative gap    = -1.62e-07
 rel. primal infeas     = 1.21e-10
 rel. dual   infeas     = 5.91e-08
 norm(X), norm(y), norm(Z) = 9.3e+00, 7.6e-01, 1.6e+00
 norm(A), norm(b), norm(C) = 1.0e+02, 1.0e+00, 1.7e+00
 Total CPU time (secs)  = 0.22
 CPU time per iteration = 0.01
 termination code       =  0
 DIMACS: 1.2e-10  0.0e+00  5.9e-08  0.0e+00  -1.6e-07  4.9e-09
------------------------------------------------------------------
ans =
  struct with fields:

    yalmipversion: '20200930'
    matlabversion: '9.7.0.1190202 (R2019b)'
       yalmiptime: 0.1234
       solvertime: 0.3019
             info: 'Successfully solved (SDPT3-4)'
          problem: 0
```

[52]:
```
double(lambda)
```

```
ans =
```

```
0.3679
```