Binormal Complex Symmetric Operators Notebook

Taylor University, MAT450 project

Functions

To run just these functions, right click on the bar on the left corresponding to “Functions,” and press “Evaluate Cells.” Many of these functions are interdependent, and so may cause issues if one attempts to use any given one without having evaluated the others.

Generic Mathematics: Contains functions Mathematica should probably have, as well as the few constants used in this notebook.
Special Matrices: Provides a way to generate various classes of matrices, given a base matrix.
My Matrix Checks: Test matrices for various properties via tests which are not listed in a paper. This section includes the widest variety of material.
Official Matrix Tests: The heart of our work: the strong angle test and the modulus test.

Generic Mathematics

```
ϵϵ = 10.^-10; (*Used to do things imprecisely,
with numerical precision. Needed for the program to run at a reasonable pace*)
```

```
DeleteCloseCases[t_] := Module[{T = Table[t⟦i⟧, {i, Length[t]}], k = 0},
   (*Given a list, remove elements less than ϵϵ apart.*)
   Do[
    k++;
    If[Length[T] ≥ k && Length[T] - i ≥ 1,
     Do[
       If[N[Abs[T⟦i⟧ - T⟦j⟧]] < N[ϵϵ],
        T = Delete[T, j];
        ];
       , {j, Length[T], i + 1, -1}];
    ];

    , {i, 1, Length[t]}];
   T
  ];

InnerProduct[u_, v_] := Dot[Conjugate[u], v];
(*⟨a|b⟩, as opposed to the plain dot product*)

StandardEigenvectors[M_] :=
  Module[{e, n, arg}, (*Returns normalized eigenvectors of M such
    that their first nonzero element is real. This is important since
    eigenvectors should be unaffected by an overall phase constant.*)
   n = Dimensions[M]⟦1⟧;
   e = Normalize /@ Eigenvectors[M];
   Do[
    arg = 1;
    Do[
     If[Abs[e⟦i, j⟧] > ϵϵ,
       arg = Exp[ⅈ Arg[e⟦i, j⟧]];
       Break[];
      ];
     , {j, 1, n}];
    e⟦i⟧ = e⟦i⟧ / arg;
    , {i, 1, Length[e]}];
   e
  ];

GetNumericalRange[m_] := Module[{M, norm, θ, k, w, Aθ, Hθ, eigval,
    eigvec, s, (*Tries (poorly) to approximate the set x†Ax/x†x for all x*)
    Kθ, pKp, eigval0, eigvec0, s0, temp,
    s1, ww,
    increment = 1 / 1000},
   M = N[m];
```

```
norm = Ceiling[Norm[M, 2]];
θ = Table[i, {i, 0, 2 π, increment}];
k = 1;
w = {};
Do[
 Aθ = Exp[-ⅈ * θ⟦i⟧ ] * M;
 Hθ = (Aθ + Aθ†) / 2;
 eigval = Re[Eigenvalues[M]];
 eigvec = Normalize /@ Eigenvectors[M];
 s = Position[eigval, Max[eigval] ];
 (*May need to decapsulate this. Currently looks like {{},{}}*)

 If[Length[s] == 1,

   AppendTo[w, (Extract[eigvec, s].M.Extract[eigvec, s]†)⟦1, 1⟧];
   (*eigvec⟦s⟧†.M.eigvec⟦s⟧;*)

   (*Print[(Extract[eigvec,s].M.Extract[eigvec,s]†)⟦1,1⟧ ];*)
   , (*else*)

   Kθ = ⅈ * (Hθ - Aθ);
   pKp = Extract[eigvec, s].Kθ.Extract[eigvec, s]†;
   (*eigvec⟦s⟧†.Kθ.eigvec⟦s⟧;*)

   eigval0 = Re[Eigenvalues[pKp]];
   eigvec0 = Normalize /@ Eigenvectors[pKp];
   s0 = Position[eigval0, Min[eigval0] ];
   temp = Extract[eigvec0, s0].pKp.Extract[eigvec0, s0]†;
   (*eigvec0⟦s0⟦1⟧ ⟧†.pKp.eigvec0⟦s0⟦1⟧ ⟧;*)
   (*eigvec0⟦s0⟦1⟧ ⟧†.eigvec⟦s⟧†.M.eigvec⟦s⟧.eigvec0⟦s0⟦1⟧ ⟧*)
   (*w⟦k⟧=temp⟦1,1⟧;*)
   AppendTo[w, temp⟦1, 1⟧];

   k++;
   s1 = Position[eigval0, Max[eigval0] ];
   temp = Extract[eigvec0, s1].pKp.Extract[eigvec0, s1]†;
   (*eigvec0⟦s1⟦1⟧ ⟧†.pKp.eigvec0⟦s1⟦1⟧ ⟧;*)
   (*eigvec0⟦s1⟦1⟧ ⟧†.eigvec⟦s⟧†.M.eigvec⟦s⟧.eigvec0⟦s1⟦1⟧ ⟧*)
   AppendTo[w, temp⟦1, 1⟧];
  ] ×
  k++
 , {i, Length[θ]}];
w
```

```
GetNumericalRadius[a_] := Module[
   (*Tries (poorly) to approximate the maximum absolute value of the set x†Ax/x†x.*)
   {A, z, w, θ, H, eigsys, k = 1, n = Dimensions[a]〚1〛, kmax = 10},

   A = N[a];
   z = {
     {Normalize[Table[Exp[ⅈ RandomReal[{0, 2 π}]], n]]}
    };
   w = {(z〚1〛.A.z〚1〛†)〚1, 1〛};
   k++;
   Do[
    AppendTo[z,
     {Normalize[
        (w〚k - 1〛 * z〚k - 1〛.A† + w〚k - 1〛* * z〚k - 1〛.A)〚1〛
      ]}
    ];
    AppendTo[w,
     (z〚k〛.A.z〚k〛†)〚1, 1〛
    ];

    If[ Abs[w〚k〛] < Abs[w〚k - 1〛],
     θ = Arg[w〚k - 1〛*];
     H = 1/2 (Exp[ⅈ θ] A + Exp[-ⅈ θ] A†);
     eigsys = Eigensystem[H];
     z〚k〛 = {Normalize[
        eigsys〚2,
          Position[eigsys〚1〛, Max[Abs[eigsys〚1〛]]] [[1, 1]]
          (*Ignore multiplicity of eigenvalues*)
         〛
       ]};
     w〚k〛 = (z〚k〛.A.z〚k〛†)〚1, 1〛;
    ];

    , kmax];
   w〚Length[w]〛
  ];
```

Special Matrices

```
PolarDecomposition[m_] := {#.#3†, #3.#2.#3†} &@@ SingularValueDecomposition[m];
(*A little bit of black magic. Returns {U,|T|}*)

NSlither[T_] := Module[{PD, t, U}, (*Returns a numerical approximation of
     the aluthge transformation of T. Doing it precisely is VERY slow.*)
    PD = PolarDecomposition[N[T]];
    t = MatrixPower[PD[[2]], 1/2];
    U = PD[[1]];
    t.U.t
  ];

NDuggal[T_] := Module[{PD}, (*Returns a numerical approximation
     of the dugggal transform of T. Again, precision is painful*)
    PD = PolarDecomposition[ N[t] ];
    N[PD[[2]]].PD[[1]]
  ];
```

My Matrix Checks

```
CheckNormality[T_] := Module[{t}, (*Is T normal?*)
    t = ConjugateTranspose[T];
    Norm[T.t - t.T, 1] < ϵϵ
  ];
CheckBinormality[T_] := Module[{t}, (*Is T binormal?*)
    t = ConjugateTranspose[T];
    Norm[T.t.t.T - t.T.T.t, 1] < ϵϵ
  ];
```

```
StrongAngleMetric[M_] := Module[{ev, n, minDiff},
    (*Gives M a score indicating how appropriate the strong angle test
     is. A very low score indicates the test could give wrong results.*)
    minDiff = 1;
    ev = Eigenvalues[M];
    n = Length[ev];
    If[Dimensions[M][[1]] > n, minDiff = 0;];
    Do[
     If[i ≠ j && Abs[ ev[[i]] - ev[[j]] ] < minDiff,
       minDiff = Abs[ ev[[i]] - ev[[j]] ];
      ];
     , {i, 1, n}, {j, 1, n}];
    minDiff
   ];

ModulusMetric[M_] := Module[{n, minDiff, sv},
    (*Gives M a score indicating how appropriate the modulus test is. A
     very low score indicates the test could give wrong results.*)
    minDiff = 1;
    sv = SingularValueList[M];
    n = Length[sv];
    If[Dimensions[M][[1]] > n, minDiff = 0;];
    Do[
     If[i ≠ j && Abs[ sv[[i]] - sv[[j]] ] < minDiff,
       minDiff = Abs[ sv[[i]] - sv[[j]] ];
      ];
     , {i, 1, n}, {j, 1, n}];
    minDiff
   ];
CSOTestability[M_] :=
  If[CheckNormality[M], 1, Max[ModulusMetric[M], StrongAngleMetric[M]]];
(*Checks whether we have the means to check a matrix for complex symmetry.*)

CheckNormaloid[M_] := (Abs[Norm[M, 2] - Max[Abs[Eigenvalues[M]]]] < ϵϵ);
(*Checks whether normaloid*)

CheckTheta[T_] :=
  Module[{t}, (*Checks whether T belongs to the fairly obscure class Θ.*)
   t = ConjugateTranspose[T];
   Norm[t.T.(T + t) - (T + t).t.T, 1] < ϵϵ
  ];
```

```
CheckCSO[M_] := Module[{distincteigen, distinctsingular, normal}, (*Returns True,
   False, or "Indeterminate" as to the complex symmetry of M*)
   normal = ConjugateTranspose[M].M == M.ConjugateTranspose[M];
   Which[
    normal,
    True,

    StrongAngleMetric[M] > ϵϵ^(1/2),
    StrongAngleTest[M],

    ModulusMetric[M] > ϵϵ^(1/2),
    ModulusTest[M],

    True,
    "Indeterminate"
   ]
  ];

CheckSpectraloid[T_] := Module[{w, r},
   (*Use this with caution. It uses a poor approximation to numerical radius,
   and so can easily return flawed results*)
   r = Max[Abs[N[Eigenvalues[T]]]];
   w = GetNumericalRadius[T];
   Which[
    NormalityCheck[T], True,
    w > r + ϵϵ, False,
    True, True (*BAD! This is not nescessarily, true—only in the limit does
      it become true. And maybe not there. But this probably works*)
   ]
  ];
```

```
CheckCentered[TT_] := Module[{T, t, oplist, checklength = 7, iscentered = True},
   (*Approximates whether TT is centered up to checklength places. That is,
   whether all of TT^i.TT*^i, TT*^i.TT^i commute for all i ≤ checklength.*)
   T = N[TT];
   t = T† ;
   oplist = Table[{}, 2 checklength];
   Do[
    oplist⟦2 * i - 1⟧ = MatrixPower[t, i].MatrixPower[T, i];
    oplist⟦2 * i⟧ = MatrixPower[T, i].MatrixPower[t, i];
    , {i, 1, checklength}];
   Do[
    If[j ≠ i && Norm[oplist⟦i⟧.oplist⟦j⟧ - oplist⟦j⟧.oplist⟦i⟧, 1] > ϵϵ,
      iscentered = False;
      Break[]
     ];
    , {i, 1, Length[oplist]}, {j, 1, Length[oplist]}];
   iscentered
  ];
```

Official Matrix Tests

```
ModulusTest[M_] := Module[{m, n, ve, ue, v, u, CSO, RHS, LHS}, (*Note that this
    function is only garanteed to work if the matrix is asymbolic*)
   If[ModulusMetric[M] < ϵϵ^1/2,
    Print["This test is invalid since not all singular values are distinct"];
    Return[];];

   m = ConjugateTranspose[M];
   n = Dimensions[M]⟦1⟧; (*Define our terms*)
   ve = Eigenvalues[M.m]; ue = Eigenvalues[m.M];
   v = StandardEigenvectors[M.m]; u = StandardEigenvectors[m.M];

   Do[ (*Rearrange eigenvectors to match the specifications of the test*)
    If[¬ i == j &&
       Abs[ ue⟦i⟧ - Conjugate[ve⟦j⟧] ] < ϵϵ
      , v⟦{i, j}⟧ = v⟦{j, i}⟧; ve⟦{i, j}⟧ = ve⟦{j, i}⟧];
    , {i, 1, n}, {j, 1, n}
   ];

   CSO = True;
   Do[ (*Here is where we differ most from the strong angle test*)
    LHS =
     InnerProduct[u⟦i⟧, v⟦j⟧] × InnerProduct[u⟦j⟧, v⟦k⟧] × InnerProduct[u⟦k⟧, v⟦i⟧];
    RHS =
     InnerProduct[u⟦i⟧, v⟦k⟧] × InnerProduct[u⟦k⟧, v⟦j⟧] × InnerProduct[u⟦j⟧, v⟦i⟧];
    If[N[Abs[LHS - RHS]] > ϵϵ, CSO = False; Return[];];
    , {i, 1, n}, {j, i + 1, n}, {k, j + 1, n}];
   Do[ (*Here is where we differ most from the strong angle test*)
    If[N[Abs[Abs[InnerProduct[u⟦i⟧, v⟦j⟧]] - Abs[InnerProduct[u⟦j⟧, v⟦i⟧]]]] > ϵϵ,
      CSO = False;
      Return[];];
    , {i, 1, n}, {j, i + 1, n}];
   CSO
  ];
```

```
StrongAngleTest[M_] := Module[{m, n, ue, ve, u, v, CSO, RHS, LHS},
   (*Only garanteed to work if the matrix is asymbolic*)
   If[StrongAngleMetric[M] < ϵϵ^1/2,
    Print["This test is invalid since not all eigenvalues are distinct"];
    Return[];];

   m = ConjugateTranspose[M];
   n = Dimensions[M][[1]]; (*Define our terms*)
   ue = Eigenvalues[M]; ve = Eigenvalues[m];
   u = StandardEigenvectors[M]; v = StandardEigenvectors[m];

   Do[ (*Rearrange eigenvectors to match the specifications of the test*)
    If[¬ i == j &&
       Abs[ ue[[i]] - Conjugate[ve[[j]]] ] < ϵϵ
     , v[[{i, j}]] = v[[{j, i}]]; ve[[{i, j}]] = ve[[{j, i}]]];
    , {i, 1, n}, {j, 1, n}
   ];

   CSO = True;
   Do[ (*Optimiztaion is vey possible here, but not needed, probably*)
    If[¬ (i == j && i == k), (*This is specified by the original paper,
      if not by Ruth's and Becca's.*)
      RHS = Conjugate[InnerProduct[v[[i]], v[[j]]] ×
         InnerProduct[v[[j]], v[[k]]] × InnerProduct[v[[k]], v[[i]]]];
      LHS = InnerProduct[u[[i]], u[[j]]] ×
        InnerProduct[u[[j]], u[[k]]] × InnerProduct[u[[k]], u[[i]]];
      If[Abs[RHS - LHS] > ϵϵ, CSO = False; Return[];];
     ];
    , {i, 1, n}, {j, i, n}, {k, j, n}];
   CSO
  ];
```

Matrix Generator

We let the matrix generator be its own section--rather than making it a function--because it is essential to be able to easily edit the matrix generator. Passing in all of the arguments to a function would only generate confusion.

To use the matrix generator:
1. Set parameters n and r below. Suggested values are 3 and 1 respectively.
2. Change the if statements inside the first cell to reflect the matrices you want to find. There are two nested if statements. While using either one alone (and letting the other always pass) would work,

having two sets of conditions lets you see how many matrices came close to being matching your conditions. This gives you an idea of how good the matrix generator is at finding things similar to what you want.
3. Evaluate the first cell.
4. Once the cell has evaluated for some time, abort it via **alt-comma**. It will not stop running by itself. The time it should run varies depending on the parameters and how rare the thing you're looking for is. Experiment here.
5. Evaluate the second cell to show total iterations, nearly valid iterations, the total number of matrices which passed, and each matrix that passed.

```
MatrixList = {};
n = 3; (*Dimensions of the matrix*)
r = 1; (*Each matrix element is selected randomly from {-r,-r+1,...,r-1,r}*)

iterates = 0;
totaliterates = 0;
Quiet[Do[
   M = N[RandomInteger[{-r, r}, {n, n}] + ⅈ RandomInteger[{-r, r}, {n, n}]];
   (*Add this onto the last bit of code to permit complex matrices*)
   totaliterates += 1;
   If[CheckTheta[M],

     iterates += 1;

     If[! CheckNormality[M], AppendTo[MatrixList, M];];
     (*If the program gets to this point, it will add M to its list*)
   ];
   , ∞]];
```

```
$Aborted
```

```
Print["The program has examined ", totaliterates, " total iterations."];
Print[iterates, " matrices passed the first test."];
Print["The following ", Length[MatrixList], " matrices passed both tests:"];
MatrixList
```

```
The program has examined 527353 total iterations.

392 matrices passed the first test.

The following 0 matrices passed both tests:

{}
```