1 Strategy

1. Compute the matrix U that diagonalizes S.
2. Compute the matrix $U_n = \mathrm{Diag}(U, ..., U)$ and $U_n^{-1} = \mathrm{Diag}(U^{-1}, ..., U^{-1})$ as change-of-basis matrices.
3. Generate SNOVA public alternating forms $Q_1, ..., Q_{ml^2}$.
4. Change the basis.
5. $\Longrightarrow$ $\mathcal{V}$ has block-diagonal form.
6. Build V as a matrix of variable with block-diagonal form.
7. Wedge V with all $Q_1, ..., Q_{ml^2}$.
8. Populate all linear equations obtained into a matrix M.
9. Compute rank of M and compare with conjectures.

2 Code

2.1 Generate a random symmetric matrix S with an irreducible characteristic polynomial

```
function RandomSymmetricIrredCharpolyMatrix(F, l : max_attempts := 10^5)
    for attempt in [1..max_attempts] do
        M := ZeroMatrix(F, l, l);
        for i in [1..l] do
            for j in [i..l] do
                a := Random(F);
                M[i][j] := a;
                M[j][i] := a;
            end for;
        end for;
        if Determinant(M) eq 0 then continue; end if;
        if IsIrreducible(CharacteristicPolynomial(M)) then
            return M;
        end if;
    end for;
    // error "No symmetric irreducible-charpoly matrix found";
end function;

S := RandomSymmetricIrredCharpolyMatrix(F, l);
```

2.2 Compute the change-of-basis matrix U such that USU^{-1} is a diagonal matrix

```
// The splitting field  K = F[]/(f)  of  S
f          := CharacteristicPolynomial(S);
K<a>   := ext<F | f>;                         //  K  GF(2^)
SK         := ChangeRing(S, K);                   //  view S in K^(×)

//  Change-of-basis matrices that diagonalise the symmetric S-matrix

function DiagonalisingBasisMatrices(S)
    l   := Nrows(S);
    assert Ncols(S) eq l;

    // 1.  The  conjugate roots of f in K are a, a^2, ..., a^(2^{l-1})
    q          := #F;
    lambdas    := [ a^(q^i) : i in [0..l-1] ];      // distinct

    // 2.  Build U column-wise from eigen-vectors
    U := ZeroMatrix(K, l, l);
    for i in [1..l] do
        z   := lambdas[i];
        V := Nullspace(SK - z*IdentityMatrix(K, l));
        v := Basis(V)[1];                   // one non-zero eigenvector

        vmat := Matrix(K, l, 1, Eltseq(v));  // make an ×1 matrix
        InsertBlock(~U, vmat, 1, i);         // insert at column i
    end for;

    return U^-1, U;
end function;

U, UInversed := DiagonalisingBasisMatrices(S);
```

2.3 Build $U_n = \textbf{Diag}(U, ..., U)$

```
// Build the change-of-basis matrix Un = Diag(U,..,U)
Un := U;
for k in [2..n] do
    Un := DiagonalJoin(Un, U);
end for;
UnInversed := UInversed;
for k in [2..n] do
    UnInversed := DiagonalJoin(UnInversed, UInversed);
end for;
```

2.4 Build SNOVA public alternating forms

```
function SnovaPublicMatrices(n, v, m, l, S)
    o := n - v;
    //---------- F_q[S] ----------
    RandFqS   := func< |
        &+[ Random(F) * S^(i-1) : i in [1..l] ] >;

    //---------- SNOVA permutation matrices z(S^j) ----------------------
    Lambda := function(Q)
        // Build block-diagonal matrix with n copies of Q
        M := Q;
        for i in [2..n] do
            M := DiagonalJoin(M, Q);
        end for;
        return M;
    end function;
    LambdaS := [ Lambda(S^(i-1)) : i in [1..l] ];

    //---------- 1. central matrix F with zero oil{oil block ------------
    RandF := function()
        F11 := RandomMatrix(F,v*l,v*l);
        F12 := RandomMatrix(F,v*l,o*l);
        F21 := RandomMatrix(F,o*l,v*l);
        ZZ  := ZeroMatrix(F, o*l, o*l);

        Row1 := HorizontalJoin(F11, F12);
        Row2 := HorizontalJoin(F21, ZZ);
        return VerticalJoin(Row1, Row2);
    end function;

    Flist := [ RandF() : k in [1..m] ];

    //---------- 2. upper-triangular hiding matrix  T -------------------
    Tvo := BlockMatrix(v, o, [ RandFqS() : k in [1..v*o] ]);
    Iv  := IdentityMatrix(F, v*l);
    Io  := IdentityMatrix(F, o*l);
    Zvo := ZeroMatrix(F, o*l, v*l);

    Row1 := HorizontalJoin(Iv,  Tvo);
    Row2 := HorizontalJoin(Zvo, Io);
    T    := VerticalJoin(Row1, Row2);

    //---------- 3. final public skew-sym. forms ------------------------
    Plist := [ Transpose(T) * Fmat * T : Fmat in Flist ];
    Qlist := [];
```

```
    for P in Plist do
        SymP := P + Transpose(P);
        for Lj in LambdaS do
            for Lk in LambdaS do
                Append(~Qlist, Lj * SymP * Lk);
            end for;
        end for;
    end for;

    return Qlist;
end function;

Q_original := SnovaPublicMatrices(n,v,m,l,S);
```

2.5 Build variable matrix V

```
////////////////////////////////////////////////////////////////////////
//   RANDOM ELEMENT  F_q[S]  (× matrix over F)
////////////////////////////////////////////////////////////////////////
RandFqS := function()
    return &+[ Random(F) * S^(i-1) : i in [1..l] ];
end function;

////////////////////////////////////////////////////////////////////////
//   POLYNOMIAL RING  R  F[x_{r,i,j}]
////////////////////////////////////////////////////////////////////////
// -----  build  R = F[x_0, ..., x_{Nvars-1}] ----- //
Nvars := v*n*l;
R     := PolynomialRing(K, Nvars);
names := [ Sprintf("x%o%o%o", r,i,j)
            : r in [0..v-1], i in [0..n-1], j in [0..l-1] ];
AssignNames(~R, names);

// helper to map (r,i,j) → generator index
Idx := function(r,i,j) return r*n*l + i*l + j + 1; end function;
Var := function(r,i,j) return R.(Idx(r,i,j)); end function;

// ////////////////////////////////////////////////////////////////////////
// //    BUILD THE BIG VARIABLE MATRIX  X (i.e. V) (v × n blocks)
// ////////////////////////////////////////////////////////////////////////

DiagonalBlock := function(r, i)
    M := ZeroMatrix(R, l, l);
    for a in [1..l] do
        M[a][a] := Var(r, i, a-1);
    end for;
```

```
    return M;
end function;

blockrows := [ [ DiagonalBlock(r-1,i-1) : i in [1..n] ] : r in [1..v] ];
rowcat    := [ HorizontalJoin(seq) : seq in blockrows ];
X         := VerticalJoin(rowcat);
```

2.6 Compute minors of X which are not zero

```
////////////////////////////////////////////////////////////////////////
//    vl×vl  MINORS  OF  X
////////////////////////////////////////////////////////////////////////
vl    := v*l;
rowsC := [1..vl];

function Minor(I)                             // I = *sequence* of col-indices
    cols := [ i+1 : i in I ];
    return Determinant( Submatrix(X, rowsC, cols) );
end function;

Minors := AssociativeArray();
for I in Subsets({0..Ln-1}, vl) do            // I = *set* of integers
    seqI          := SetToSequence(I);       // keep order
    Minors[I]     := Minor(seqI);
end for;

mvals   := [ Minors[K] : K in Keys(Minors) ];
nonzero := #[ v : v in mvals | v ne 0 ];
printf "%o / %o minors are non-zero\n", nonzero, #Minors;
```

2.7 Build the equations in V ∧ Q

```
////////////////////////////////////////////////////////////////////////
//    BUILD THE EQUATIONS  (wedging with each Q)
////////////////////////////////////////////////////////////////////////
eqns := [];
for QQ in Qlist do
    for J in Subsets({0..Ln-1}, vl+2) do
        coeff := R!0;
        seqJ  := SetToSequence(J);                   // ascending order

        for aidx in [1..#seqJ-1] do
            for bidx in [aidx+1..#seqJ] do
                a := seqJ[aidx];
                b := seqJ[bidx];
```

```
                sign  := (-1)^(bidx - aidx - 1);

                // columns left after removing a and b
                Iseq  := [ seqJ[k] : k in [1..#seqJ] | k ne aidx and k ne bidx ];
                Iset  := SequenceToSet(Iseq);

                minor := Minors[Iset];
                coeff +:= sign * QQ[a+1][b+1] * minor;
            end for;
        end for;

        Append(~eqns, coeff);
    end for;
end for;
printf "Generated %o equations\n", #eqns;
```

2.8 Build the Macaulay matrix

```
 ////////////////////////////////////////////////////////////////////////
//    MACAULAY MATRIX  (target degree  = v*l)
////////////////////////////////////////////////////////////////////////

// 1. Prepare an associative array to give each monomial a tiny \column index"
mon2col := AssociativeArray();
nextCol := 1;

// 2. Start with an empty sparse matrix over F
M := SparseMatrix(K);    // 0×0 to start

// 3. Walk through the equations, assigning columns on the fly
interval := Max(1, Floor(#eqns/100));
startCPU := Cputime();

for r in [1..#eqns] do
    poly := eqns[r];
    cfs  := Coefficients(poly);
    ms   := Monomials(poly);
    for k in [1..#ms] do
        E := Exponents(ms[k]);   // exponent sequence of length Nvars

        // if we haven't yet seen this exact monomial, give it the next column
        if not IsDefined(mon2col, E) then
            mon2col[E] := nextCol;
            nextCol +:= 1;
        end if;
```

```
        // insert the coefficient into row r, that (new) column
        SetEntry(~M, r, mon2col[E], cfs[k]);
    end for;
    // | report progress every `interval` iterations, and at the end |
    if r mod interval eq 0 or r eq #eqns then
        totalSec := Cputime() - startCPU;                 // Real
        pct      := Floor(100 * r / #eqns);               // integer %
        elapsed  := Floor(totalSec);                      // integer seconds
        eta      := r gt 0
                select Floor((#eqns - r) * totalSec / r)
                else 0;                                // integer seconds

        printf
        "Processed %o/%o (%o%%)  elapsed: %o s  ETA: %o s\n",
        r, #eqns, pct, elapsed, eta;
    end if;
end for;
```

2.9 Compute rank of M and compare with hypotheses

```
////////////////////////////////////////////////////////////////////////
//    Hypotheses
////////////////////////////////////////////////////////////////////////

///////////////////////////////////////////////////////////////// Hyp1
function Hyp1(m,v,o,l)
    sum := 0;
    out  := [];
    up   := Floor(o*l/2) + 1;
    for i in [1..up] do
        coeff := (-1)^i * Bin(m*l + i - 1, i);
        inner := 0;
        for vec in NonNegVectors(2*i, l) do
            inner +:= Prod([ Bin(v+o, v+aj) : aj in vec ]);
        end for;
        sum -:= coeff*inner;
        Append(~out, <i, sum>);
    end for;
    return out;
end function;

///////////////////////////////////////////////////////////////// Hyp2
function Hyp2(m,v,o,l)
    sum := 0;
    out  := [];
    up   := Floor(o*l/2) + 1;
```

```
    for i in [1..up] do
        coeff := (-1)^i * Bin(m + i - 1, i);
        inner := 0;
        for vec in NonNegVectors(2*i, l) do
            inner +:= Prod([ Bin(v+o, v+aj) : aj in vec ]);
        end for;
        sum -:= coeff*inner;
        Append(~out, <i, sum>);
    end for;
    return out;
end function;

////////////////////////////////////////////////////////////////// Hyp3
function Hyp3(m,v,o,l)
    return [ Binomial(l*n, v) -
            &+[ (-1)^i * Bin(m*l*l + i - 1, i) * Bin(v+o, v+2*i)
                : i in [0..t] ]
        : t in [0..Floor(o/2)] ];
end function;

printf "Hypothesis 1 (Ray's l-scaled formula): %o\n", Hyp1(m,v,o,l);

printf "Hypothesis 2 (Ray's no-l formula): %o\n", Hyp2(m,v,o,l);

printf "Hypothesis 3 (plain UOV formula): %o\n", Hyp3(m, l*v, l*o, l)[#Hyp3(m, l*v, l*o,
```

3 Check $l = 1$

In theory, Rank$M$ = Min(1,