APPENDIX: BORDER APOLARITY IMPLEMENTATION

1. Introduction

Given a tensor $T \in A \otimes B \otimes C$, a natural number r, and a reductive group $G_T \subset \mathrm{GL}(A) \times \mathrm{GL}(B) \times \mathrm{GL}(C)$ stabilizing T, this file implements complete enumeration of candidate subspaces $F_{110} \subset T(C^*)^\perp \subset A^* \otimes B^*$ which (i) have codimension r in $A^* \otimes B^*$, (ii) are fixed under the Borel $\mathbb{B}_T$ of G, and (iii) which pass the (210) and (120) tests. That is, for such F_{110}, the symmetrization maps

$$F_{110} \otimes A^* \to S^2 A^* \otimes B^* \text{ and} \tag{1}$$

$$F_{110} \otimes B^* \to A^* \otimes S^2 B^* \tag{2}$$

have image of codimension at least r. More precisely, we enumerate the matrices representing such subspaces F_{110} in fixed bases. Such candidates may occur in positive dimensional families, so these matrices may have entries in the coordinate ring of some affine variety.

The same computation with the roles of A, B, and C permuted may be performed to enumerate the candidate F_{101} and F_{011} subspaces. Given a triple of candidates $(F_{110}, F_{101}, F_{011})$, a routine to apply the (111) test is provided. That is, the map

$$F_{110} \otimes C^* \oplus F_{101} \otimes B^* \oplus F_{011} \otimes A^* \to A^* \otimes B^* \otimes C^* \tag{3}$$

is implemented, and it may be checked if the codimension of its image is at least r.

1.1. Input description.

Choose bases of A, B and C consisting of weight vectors under the torus of G_T. The tensor T is described by its coefficients with respect to these bases. The action of $\mathbb{B}_T$ is described by the weights of the basis vectors along with the representations $\mathfrak{n} \to \mathfrak{gl}(A)$, $\mathfrak{n} \to \mathfrak{gl}(B)$, and $\mathfrak{n} \to \mathfrak{gl}(C)$. Concretely, representations of $\mathfrak{n}$ are described by giving the matrix of the action of the simple root vectors in the distinguished bases.

1.2. Software used.

The code is written in Python 3, using the libraries provided by SageMath. It has been tested with SageMath 9.5.

2. Program Listing

```
from sage.all import *
from itertools import product,combinations
```

2.1. Top level routines.

2.1.1. *border_apolarity_cycl_inv.* Apply the ideal enumeration algorithm up to the (111) test, assuming T has cyclic symmetry. Cyclic symmetry is used to avoid the computation of the (101) and (011) candidates.

```
def border_apolarity_cycl_inv(T,reps,r,verbose=True):
    # Check that the data of reps actually stabilize T and compute the simple roots
    simple_roots = check_T_is_stabilized(T,reps)
    # Find the representation data for T(C*)^⊥ and report the matrix of
    # the embedding into A* ⊗ B*
    reptp,em = Tperp_rep_and_embedding(T,reps)
    # Find the weight diagram of T(C*)^⊥
    wtd = weight_diagram(reptp,simple_roots)

    a = len(T)
    b = T[0].nrows()
    cand110 = []

    # Enumerate Borel fixed subspace families of T(C*)^⊥. These are given
    # by matrices Stperp over a polynomial ring and a corresponding ideal I the
```

```
    # coefficients of Stperp are considered modulo, i.e., the family takes
    # parameters in V(I).
    for Stperp,I in borel_fixed_subspaces(wtd, em.nrows()-r, verbose):
        # Embed into A^* ⊗ B^*
        Sab = em*Stperp

        # Restrict to the closed subset of V(I) passing the (210) test
        S210 = matrix_11_to_21(Sab,a)
        I = minors_ideal(S210, S210.nrows() - r + 1, I)

        # Restrict to the closed subset of V(I) passing the (120) test
        S120 = matrix_11_to_21(transpose_tensor(Sab,a),b)
        I = minors_ideal(S120, S120.nrows() - r + 1, I)

        # Check if the family is nonempty
        if QQ(1) not in I:
            # If there is a single solution in the family add it to cand110,
            # otherwise raise an error. This is sufficient for M_⟨3⟩ and det_3.
            cand110.append(Sab.apply_map(lambda e: QQ(I.reduce(e)), QQ))

    cand111 = []
    for xs in product(*map(enumerate,[cand110]*3)):
        ixs = tuple(i for i,x in xs)
        if ixs != min([ixs,ixs[1:]+ixs[:1],ixs[2:]+ixs[:2]]):
            # Skip triples equal to others we check modulo cyclic permutation
            continue
        if verbose:
            print(ixs,end=' ')
            sys.stdout.flush()

        W = matrix_to_111(*[x for _,x in xs])
        if W.rank() <= W.nrows() - r:
            cand111.append(W)
    return cand111,cand110
```

2.1.2. *check_T_is_stabilized.* Given $T \in A \otimes B \otimes C$ and the representations of A, B, and C, this function checks the input data for correctness in various ways (see the inline notes), and also computes the simple roots implied by the representation data.

```
def check_T_is_stabilized(T,reps):
    a = len(T)
    b,c = T[0].dimensions()

    # Check there is one weight for each basis vector
    assert all(len(wts) == d for d,(wts,_) in zip((a,b,c),reps))
    # Check the given representations of n operate on the correct dimensional space
    assert all(x.nrows() == d and x.ncols() == d
            for d,(_,xs) in zip((a,b,c),reps) for x in xs)

    # Compute roots of the simple roots vectors xs
    simple_roots = []
    for xi in range(len(reps[0][1])):
        for wts,xs in reps:
            if not xs[xi].is_zero():
                i,j = xs[xi].nonzero_positions()[0]
                simple_roots.append(wts[i] - wts[j])
                break

    # Check xs are actally root vectors for the corresponding roots and that
    # the simple roots agree for each rep in reps
```

```
    for wts,xs in reps:
        assert len(simple_roots) == len(xs)
        for root,x in zip(simple_roots,xs):
            for i,j in x.nonzero_positions():
                assert wts[i]-wts[j] == root

    # Check T is weight zero in A⊗B⊗C
    for i,m in enumerate(T):
        for j,k in m.nonzero_positions():
            assert (reps[0][0][i] + reps[1][0][j] + reps[2][0][k]).is_zero()

    # Check T is closed under π
    (_,xsA), (_,xsB), (_,xsC) = reps
    for xa,xb,xc in zip(xsA, xsB, xsC):
        xaT = [sum(xa[i,j] * T[j] for j in range(a)) for i in range(a)]
        xbT = [xb*m for m in T]
        xcT = [m*xc.T for m in T]
        # we should have xaT + xbT + xcT = 0
        assert all((m1+m2+m3).is_zero() for m1,m2,m3 in zip(xaT, xbT, xcT))

    return simple_roots
```

2.2. **$\mathbb{B}$-fixed subspace enumeration.** For a module M, $\mathbb{B}$-fixed subspaces are parameterized by a choices of subspaces $S_\lambda \subset M_\lambda$ where $x.S_\lambda \subset S_\mu$ for every raising operator x corresponding to an arrow $M_\lambda \to M_\mu$ in the weight diagram (see §2.5.2 of the article). This parameterization is composed of two components, the combinatorial data $d_\lambda = \dim S_\lambda$, and the subvariety Y_{d_λ} of the product of Grassmannians $X_{d_\lambda} = \prod_\lambda G(d_\lambda, M_\lambda)$ corresponding to the condition on weight diagram arrows above.

For fixed D, most assignments $\lambda \mapsto d_\lambda$ satisfying $\sum_\lambda d_\lambda = D$ have $Y_{d_\lambda} = \varnothing$, and to explicitly check each such d_λ would be impossible. Hence, we investigate only such assignments d_λ satisfying necessary conditions for $Y_d \neq \varnothing$. These conditions take the form of linear inequalities. Thus, the assignments we explicitly consider occur as the set of integer points of a polytope, which can be efficiently enumerated. The formation of this polytope and the enumeration of its integer points is done by `possible_dim_assignments` (§2.2.3), and the parameterization of Y_{d_λ} is performed by `borel_fixed_subspaces_dlambda` (§2.2.2). The two elements of the enumeration are combined in `borel_fixed_subspaces` (2.2.1).

2.2.1. *`borel_fixed_subspaces`.* Given a weight diagram `wtd` and an integer D, this function returns a generator yielding a set of families of $\mathbb{B}$-fixed subspaces of dimension D of the module corresponding to `wtd` which together exhaust all such subspaces. Each family is given as pair of (i) a matrix with coefficients in a polynomial ring and (ii) an ideal I of the coefficient ring. The coefficients of the matrix should be interpreted mod I, i.e., the matrix should be interpreted as a family of matrices over $\mathbb{C}$ parameterized by $V(I)$. The image of such a matrix is the corresponding $\mathbb{B}$-fixed subspace.

```
def borel_fixed_subspaces(wtd,D,verbose=False):
    for i,dlambda in enumerate(possible_dim_assignments(wtd,D)):
        if verbose:
            print(i,end=' ')
        for S in borel_fixed_subspaces_dlambda(wtd,dlambda,verbose):
            yield S
```

2.2.2. *`borel_fixed_subspaces_dlambda`.* Given a weight diagram and an assignment $\lambda \mapsto d_\lambda$, this function enumerates matrices corresponding to Y_{d_λ}.

A Grassmannian $G(s, V)$ may be represented in bases as $s \times \mathbf{v}$ matrix modulo the left action of GL_s. A normal form for this action is ordinary echelon form of the matrix, so $G(s, V)$ may be written as a disjoint union of affine spaces according to the pivot columns of echelon forms.

Thus, the set X_{d_λ} is parameterized by combinations of such choices of pivot columns in each weight space. This function looks at each of these and computes the ideal of equations corresponding to the intersection with Y_{d_λ}. A list of matrices with polynomial entries and corresponding ideals is returned.

```
def borel_fixed_subspaces_dlambda(wtd,dlambda,verbose=False):

    dlist = [(wt,d,len(wtd.get_vertex(wt))) for wt, d in dlambda.items()]

    if verbose:
        missing = [wt for wt,d in dlambda.items() if d < len(wtd.get_vertex(wt))]
        missing.sort(key=lambda wt: (sum(wt),wt))
        print('missing',missing)


    if all(m == f or m == 0 for wt,m,f in dlist):
        # S is full in full in every weight space for which it is nonzero.
        # Such a set satisfies the weight diagram arrow conditions due to the
        # necessary conditions on d_λ. No need to check again
        yield (identity_matrix(QQ, sum(f for _,_,f in dlist))[:,
                [i for wt,m,f in dlist if m == f for i in wtd.get_vertex(wt)]],QQ.ideal(0))
        return

    # Otherwise, we need parameters. Here we check all combinations of choices
    # of pivot columns in each grassmannian
    for nzsi,nzs in enumerate(product(*[combinations(range(f),m) for wt,m,f in dlist])):
        if verbose:
            print(nzsi,end=' ')
            sys.stdout.flush()

        nvars = sum( (nzi+1)*(j-i-1) for (wt,m,f),nz in zip(dlist,nzs)
            for nzi,(i,j) in enumerate(zip(nz,nz[1:]+(f,))))
        R = PolynomialRing(QQ,'t',nvars,implementation='singular') if nvars > 0 else QQ

        S = {}
        xi = 0
        for (wt,m,f),nz in zip(dlist,nzs):
            t = matrix(R,f,m,sparse=True)
            t[nz,:] = identity_matrix(R,m,sparse=True)
            for j,ks in enumerate(zip(nz,nz[1:]+(f,))):
                inc = (ks[1]-ks[0]-1)*(j+1)
                t[ks[0]+1:ks[1],:j+1] = matrix(R,ks[1]-ks[0]-1,j+1,R.gens()[xi:xi+inc])
                xi += inc

            S[wt] = (t, nz) # remember the pivots for equations below

        # Now restrict the parameters appearing so that S is closed under the
        # arrows of the weight diagram
        eqs = []
        for wta,wtb,x in wtd.edges():
            S1,_ = S[wta]
            S2,S2pvts = S[wtb]
            # Reduce the columns of xS1 modulo S2 using the pivots of S2.
            xS1 = (x*S1).T
            S2 = S2.T
            for i,j in enumerate(S2pvts):
                for k in xS1.nonzero_positions_in_column(j):
                    xS1[k] -= xS1[k,j]*S2[i]
            # xS1 now is in a normal form modulo S2, so the equations of
            # containment are the conditions that xS1 == 0
            eqs.extend(xS1.dict().values())

        I = R.ideal(eqs)
        if R.one() in I:
```

```
            continue

        Smat = block_diagonal_matrix([S[wt][0] for wt,_,_ in dlist],sparse=True)

        ix = [i for wt,_,_ in dlist for i in wtd.get_vertex(wt)]
        ixi = [None]*len(ix)
        for j,i in enumerate(ix):
            ixi[i] = j
        Smat = Smat[ixi,:]

        yield (Smat,I)

    if verbose:
        print()
```

2.2.3. *possible_dim_assignments.* Given a weight diagram `wtd` and a target dimension `dim`, this function computes the integer points d_λ of the polytope corresponding to the following necessary conditions that assignments d_λ have $Y_{d_\lambda} \neq \varnothing$ (and $\sum_\lambda d_\lambda =$ `dim`):

(1) Suppose that $M_\mu \to M_{\lambda_i}$ in the weight diagram, where λ_i ranges over some set of weights. The map $y : M_\mu \to \bigoplus_i M_{\lambda_i}$ satisfies $y(S_\mu) \subset \bigoplus_i S_{\lambda_i}$. In particular, $d_\mu \leq \dim\ker y + \sum_i d_{\lambda_i}$.
(2) Dually, suppose that $M_{\mu_i} \to M_\lambda$ in the weight diagram, where μ_i ranges over some set of weights. The map $z : M_\lambda^* \to \bigoplus_i M_{\mu_i}^*$ satisfies $z(S_\lambda^\perp) \subset \bigoplus_i S_{\mu_i}^\perp$. In particular, $\dim M_\lambda - d_\lambda \leq \dim\ker z + \sum_i \dim(M_{\mu_i}) - d_{\mu_i}$.
(3) One can apply the previous conditions to compositions of arrows in the weight diagram. We use only the condition corresponding to a single arrow $w : M_\mu \to M_\nu \to M_\lambda$. For a single arrow, conditions (1) and (2) are the same: $d_\mu \leq \ker\dim w + d_\lambda$.

```
def possible_dim_assignments(wtd,dim):
    lp = MixedIntegerLinearProgram()
    for wt in wtd:
        lp.set_min(lp[wt],0)
        lp.set_max(lp[wt],len(wtd.get_vertex(wt)))

    lp.add_constraint(lp.sum(lp[wt] for wt in wtd) == dim)

    for wt in wtd:
        # Condition (1)
        for k in range(1,len(wtd.outgoing_edges(wt))+1):
            for es in combinations(wtd.outgoing_edges(wt),k):
                kdim = block_matrix([[xr] for _,_,xr in es]).right_kernel().dimension()
                lp.add_constraint(lp[wt] <= kdim + lp.sum(lp[wtr] for _,wtr,_ in es))

        # Condition (2)
        # This when k = 1 occurs also for condition (1) as well, so we skip it.
        for k in range(2,len(wtd.incoming_edges(wt))+1):
            for es in combinations(wtd.incoming_edges(wt),k):
                cokdim = block_matrix([[-xr.transpose()]
                    for _,_,xr in es]).right_kernel().dimension()
                lp.add_constraint(len(wtd.get_vertex(wt)) - lp[wt] <= cokdim + lp.sum(
                    len(wtd.get_vertex(wtl)) - lp[wtl] for wtl,_,xr in es))

    # Condition (3)
    for wtstart in wtd:
        for _,wtmid,x1 in wtd.outgoing_edges(wtstart):
            for _,wtlast,x2 in wtd.outgoing_edges(wtmid):
                kdim = (x2*x1).right_kernel().dimension()
                lp.add_constraint(lp[wtstart] <= kdim + lp[wtlast])

    # We have computed the polytope, now enumerate the integer points
```

```
    # Fix an ordering of the weights λ_i
    wts = wtd.topological_sort()

    # Given assignments of d_{λ_j}, j ≤ i−1, set d_{λ_i} to the values for which the
    # polytope still intersects the coordinate hyperplane corresponding to the
    # current partial assignment and recursively apply this procedure.
    from sage.numerical.mip import MIPSolverException
    def dfs(i):
        if i == len(wts):
            # We have set all coordinates to integer values and remain in the
            # polytope, report this as a solution
            yield {wt : int(lp.get_min(lp[wt])) for wt in wts}
            return
        wt = wts[i]
        mult = int(lp.get_max(lp[wt]))
        for dcur in range(0, mult+1):
            lp.set_min(lp[wt],dcur)
            lp.set_max(lp[wt],dcur)
            try:
                # The polytope has nonzero intersection with set corresponding
                # to fixing the first i coordinates as we have. Recursively try
                # to set the remaining coordinates and report all solutions.
                lp.solve()
                for pt in dfs(i+1):
                    yield pt
            except MIPSolverException:
                # There are no points in the polytope with the first i
                # coordinates fixed to the values we have used. Don't search
                # further.
                pass
        lp.set_min(lp[wt],0)
        lp.set_max(lp[wt],mult)

    return dfs(0)
```

2.3. (210) and (120) symmetrization maps, (111) addition map.

2.3.1. *matrix_11_to_21.* If B is the $\mathbf{ab}\times\mathbf{v}$ matrix of a map $V\to A\otimes B$ with respect to an lexicographically ordered basis of $A\otimes B$ of the form $a_i\otimes b_j$, this routine computes the matrix of the map $V\otimes A\to S^2A\otimes B$. The bases of the tensor products are also lexicographically ordered tensor products. The basis a_ia_j, $i\le j$, of S^2A is ordered reverse lexicographically: $a_{i_1}a_{j_1}$ comes before $a_{i_2}a_{j_2}$ if either $j_1<j_2$ or $j_1=j_2$ and $i_1<i_2$.

```
def matrix_11_to_21(B,a):
    b = B.nrows() // a
    S = B.ncols()
    W = {}
    # i,k < a
    # j < b
    # s < S
    for I,s in B.nonzero_positions():
        i,j = I // b, I % b
        v = B[I,s]
        for k in range(a):
            mi,ma = min(i,k),max(i,k)
            ix = (( binomial(ma+1,2)+mi )*b+j,s*a+k)
            W[ix] = W.get(ix,0) + v
    W = matrix(B.base_ring(),binomial(a+1,2)*b,S*a,W)
    return W
```

2.3.2. *matrix_to_111.* Suppose A,B, and C are $\mathbf{bc}\times\mathbf{s}$, $\mathbf{ca}\times\mathbf{u}$, and $\mathbf{ab}\times\mathbf{v}$ matrices corresponding to maps $S\to B\otimes C$, $U\to C\otimes A$, $V\to A\otimes B$, respectively. It is assumed that all tensor products are given with respect to lexicograpcally ordered product bases. This routine computes the matrix corresponding to the addition map $A\otimes S\oplus B\otimes U\oplus C\otimes V\to A\otimes B\otimes C$. Here tensor products are given by lexicographically ordered product bases, and the basis of the direct sum is the concatenation of the bases of the summands.

```
def matrix_to_111(A,B,C):
    a = int(sqrt(B.nrows()*C.nrows()/A.nrows()))
    b = C.nrows() // a
    c = B.nrows() // a
    W = {}
    for i in range(a):
        for I,l in A.nonzero_positions():
            j,k = I // c, I % c
            W[((i*b+j)*c+k, i*A.ncols()+l)] = A[j*c+k,l]
    for j in range(b):
        for I,l in B.nonzero_positions():
            k,i = I // a, I % a
            W[((i*b+j)*c+k, a*A.ncols() + j*B.ncols()+l)] = B[k*a+i,l]
    for k in range(c):
        for I,l in C.nonzero_positions():
            i,j = I // b, I % b
            W[((i*b+j)*c+k, a*A.ncols() + b*B.ncols() +\
                    k*C.ncols()+l)] = C[i*b+j,l]
    W = matrix(A.base_ring(),a*b*c,a*A.ncols()+b*B.ncols()+c*C.ncols(),W)
    return W
```

2.3.3. *transpose_tensor.* Given an $\mathbf{ab}\times\mathbf{s}$ matrix representing a map $S\to A\otimes B$, returns the matrix of the same size corresponding to the map $S\to B\otimes A$. Bases of tensor products are assumed to be lexicographically ordered product bases.

```
def transpose_tensor(B,a):
    b = B.nrows() // a
    S = B.ncols()
    Bp = {}
    for I,k in B.nonzero_positions():
        i,j = I // b, I % b
        Bp[(j*a+i,k)] = B[I,k]
    return matrix(B.base_ring(),B.nrows(),B.ncols(),Bp)
```

2.4. **Representation manipulation.**

2.4.1. *Tperp_rep_and_embedding.* Given $T\in A\otimes B\otimes C$ and the representations of A, B, and C, this function computes the representation of $T(C^*)^\perp\subset A^*\otimes B^*$ and the matrix of the embedding of the distinguished weight bases of these modules. Optionally, the parameter `missing` may be set to 0 to instead compute this information for $T(A^*)^\perp\subset B^*\otimes C^*$ or to 1 to do this for $T(B^*)^\perp\subset C^*\otimes A^*$.

```
def Tperp_rep_and_embedding(T,reps,missing=2):
    reps = reps[missing+1:] + reps[:missing]
    repAB = module_product(*[module_dual(rep) for rep in reps])

    Tcycl = T
    for i in range(missing):
        Tcycl = tensor_cycl(Tcycl)
    Tflattening = matrix(QQ,[m.list() for m in Tcycl],sparse=True)
    em = Tflattening.right_kernel_matrix().transpose().sparse_matrix()

    reptp = submodule_from_basis(repAB,em)
    return reptp,em
```

2.4.2. *module_product.* When `repa` and `repb` describe the representations of V, and W, respectively this function computes description of the representation of $V \otimes W$. The distinguished weight basis of the tensor product is $v_i \otimes w_j$ ordered lexicographically, where w_i and w_j are the distinguished weight bases of V and W, respectively.

```
def module_product(repa,repb):
    wtsa, xsa = repa
    wtsb, xsb = repb
    wtsab = [wta + wtb for wta,wtb in product(wtsa,wtsb)]
    xsab = []
    for xa,xb in zip(xsa,xsb):
        xab = xa.tensor_product(identity_matrix(QQ,xb.nrows()))
        xab += identity_matrix(QQ,xa.nrows()).tensor_product(xb)
        xsab.append(xab)
    return (wtsab,xsab)
```

2.4.3. *module_dual.* When `rep` describe the representation of V, this function computes the description of the representation V^*. The distinguished weight basis of V^* is the dual basis of that of V.

```
def module_dual(rep):
    wts, xs = rep
    wtsd = [-wt for wt in wts]
    xsd = [-x.transpose() for x in xs]
    return (wtsd,xsd)
```

2.4.4. *submodule_from_basis.* Here, `rep` describes the representation of V, and `em` is an $\mathbf{v} \times \mathbf{w}$ matrix containing as columns a weight basis of a submodule W of V, e.g., it is the matrix of the embedding $W \to V$. This function computes the description of the representation W with respect to this basis. The facts that B describes a weight basis and that W is a $\mathbb{B}$-submodule of V are checked.

```
def submodule_from_basis(rep,em):
    wts, xs = rep

    # check em describes a basis
    assert em.rank() == em.ncols()

    wtsw = []
    # check the columns of em are weight vectors and find their weights
    for v in em.columns():
        wt = wts[v.nonzero_positions()[0]]
        assert all(wt == wts[j] for j in v.nonzero_positions())
        wtsw.append(wt)

    xsw = []
    # check W is fixed under the simple root vectors and compute the matrix of
    # their action
    for x in xs:
        # find y solving x*em == em*y, and raises an error if there is none
        y = em.solve_right(x*em)
        # Since em is basis, the eqution x*em == em*y is sufficient to
        # guarantee the claims
        xsw.append(y)

    return (wtsw,xsw)
```

2.4.5. *weight_diagram.* Given a representation `rep` and the set of simple roots consistent with it, this routine computes the corresponding weight diagram, a directed graph with vertices labelled by weights and with edges labelled with the matrix of the restriction of the corresponding raising operator. Each vertex records the ordered list of corresponding distinguished weight basis vectors (with respect to which the edge matrices are given).

```
def weight_diagram(rep,simple_roots):
    # tot = simultaneous_eigenspace(a[2])
    wts, xs = rep
    dim = len(wts)

    assert all(x.nrows() == dim and x.ncols() == dim for x in xs)

    wtd = DiGraph()
    for bi,wt in enumerate(wts):
        wt.set_immutable()
        if wt not in wtd:
            wtd.add_vertex(wt)
            wtd.set_vertex(wt,[])
        wtd.get_vertex(wt).append(bi)

    for wt in wtd:
        wt.set_immutable()
        for root,x in zip(simple_roots,xs):
            wt_raised = wt + root
            wt_raised.set_immutable()
            if wt_raised in wtd:
                wtd.add_edge(wt,wt_raised,
                        x[wtd.get_vertex(wt_raised),wtd.get_vertex(wt)])

    return wtd
```

2.5. **Efficiently computing the ideal of $r \times r$ minors.**

2.5.1. *minors_ideal.* Given an integer r and a matrix m with entries in a polynomial ring interepreted modulo the ideal I, this function computes an ideal $J \supset I$ which set theoretically cuts out the closed set X on which m has rank at most $r-1$. That is, if I_r is the ideal generated by the $r \times r$ minors of m, then $I_r + I \subset J \subset \sqrt{I_r + I}$.

This routine attempts to compute such a J without enumerating all of the $r \times r$ minors of m, which may be cost prohibitive. The algorithm used is essentially row reduction by units modulo I. When such reduction cannot proceed, a nonzero coefficient f of the matrix is heuristically selected and the algorithm recursively analyzes two cases according to the decomposition $X = (X \cap V(f)) \cup (X \setminus V(f))$. Algebraicly, the first case corresponds to continuing the computation with I replaced with $I + (f)$, and the second case corresponds to continuing the computation in the polynomial ring localized at f. If S is the polynomial ring over which m and I are defined, localization is implemented by working in the ring $R = S[d]$, d an additional indeterminant, with I replaced by $I + (df - 1)$.

Since the algorithm is recursive, all computations are done in the ring R and the current element d_0 associated to the inverse of d is remembered. When it is needed to localize with respect to an additional element f, d is reinterpreted as the inverse of $d_0 f$ by substuting d with df in the enteries of m and replacing I with $I \cap S + (dd_0 f - 1)$.

```
def minors_ideal(m,r,I=None):
    from collections import Counter

    S = m.base_ring()
    if I is None: I = S.ideal()

    if S is QQ:
        return QQ.ideal(1) if QQ(1) in I or m.rank() >= r else QQ.ideal(0)

    R = PolynomialRing(S.base_ring(),S.gens()+('d',))
    dv = R.gens()[-1]

    def rec(m,I,d,r):
        if m.is_zero():
```

```
            return I.elimination_ideal(dv).change_ring(S)
        if r == 1:
            return (I + m.coefficients()).elimination_ideal(dv).change_ring(S)

        # select the most common nonzero element of
        f = Counter(m.dict().values()).most_common(1)[0][0]

        # Case 1: pass to the closed set where f == 0
        Icur = I + f
        mcur,lo = elimination_by_units(m,Icur)
        I1 = rec(mcur[lo:,lo:],Icur,d,r-lo) if lo < r else S.ideal(1)

        # Case 2: pass to the open set where f not identically zero, e.g., localize at f
        mcur = m.subs({dv : dv*f})
        Icur = I.elimination_ideal(dv) + [dv*d*f-1]
        mcur,lo = elimination_by_units(mcur,Icur)
        I2 = rec(mcur[lo:,lo:],Icur,d*f,r-lo) if lo < r else S.ideal(1)

        # Combine the result of both cases: I1 ∩ I2 corresponds to the
        # union of the corresponding parameter sets
        return I1.intersection(I2)

    return rec(m.change_ring(R), I.change_ring(R), R.one(), r)
```

2.5.2. *elimination_by_units.* This routine implements row elimination by unit coefficients of m mod I with column pivoting. It returns the resulting matrix m' and a number r. m' has the form

$$\begin{bmatrix} T & A \\ 0 & B \end{bmatrix},$$

where T is $r \times r$, upper triangular, with ones on the diagonal, and B contains no unit coefficients.

```
def elimination_by_units(m,I):
    m = m.apply_map(lambda e: I.reduce(e))

    from sage.libs.singular.function import singular_function
    lift = singular_function('lift')

    # computes the inverse mod I if it exists, otherwise None
    def try_inverse(e):
        try:
            return lift(I+e,1).list()[-1]
        except RuntimeError:
            return None

    r = 0
    while True:
        if r == min(m.nrows(),m.ncols()):
            break

        einv = None
        for (i,j),e in m[r:,r:].dict().items():
            einv = try_inverse(e)
            if einv is not None:
                break
        if einv is None:
            break
        i += r
        j += r

        m.swap_rows(r,i)
```

```
            m.swap_columns(r,j)
            m[r,:] *= einv
            for k in m.nonzero_positions_in_row(r):
                m[r,k] = I.reduce(m[r,k])
            assert m[r,r] == 1
            for i in m.column(r)[r+1:].nonzero_positions():
                i += r+1
                m.add_multiple_of_row(i,r,-m[i,r])
                for k in m.nonzero_positions_in_row(i):
                    m[i,k] = I.reduce(m[i,k])
            r += 1

    return m,r
```

2.6. **Miscellaneous.**

2.6.1. *tensor_cycl.* When $T \in A \otimes B \otimes C$, this function cyclicly permutes the factors to obtain a matrix $T \in B \otimes C \otimes A$.

```
def tensor_cycl(T):
    m = len(T)
    n,s = T[0].dimensions()
    S = [zero_matrix(QQ,s,m) for i in range(n)]
    for i,j,k in product(range(m),range(n),range(s)):
        S[j][k,i] = T[i][j,k]
    return S
```