# Documentation for GHilb4orbifolds script

Sara Muhvić
University of Warwick

November 18, 2017

## Contents

Here we present the code used for computing all the torus invariant $G$-clusters, for the group $G = (\mathbb{Z}/r)^{\oplus n-1}$ acting on $\mathbb{C}^n$ by (??). The code is written in Sage [1]. At the end we show the basic usage of the class.

## 1 The `SymQuotSing` class

The main class is called `SymQuotSing` and it represents the quotient variety $\mathbb{C}^n/G$. An object of type `SymQuotSing` is initialized by two variables: the exponent of the group `r` and the dimension `dim` $= n$ of the affine space. During the initialization, a variable storing the order of the group is created, as well as the polynomial ring $\mathbb{C}[x_1, x_2, \ldots, x_n]$. For clarity, in dimensions up to five, the variables have names $x, y, z, t, w$ instead of $x_i$. The code uses the packages `os.path` for accessing the file tree, `cPickle` for storing the computed data in the memory, and `random` that improves the output of `__relations` method, which have to be imported prior to running the script.

```
class SymQuotSing(object):
   def __init__(self, r, dim=4):
      self.r = r
      self.dim = dim
      self.ord = r**(dim-1)
      self.__L = ZZ**self.dim

      if self.dim == 2:
         self.__Q = PolynomialRing(QQ, 2, 'xy')
      elif(self.dim == 3):
         self.__Q = PolynomialRing(QQ, 3, 'xyz')
      elif(self.dim == 4):
         self.__Q = PolynomialRing(QQ, 4, 'xyzt')
      elif(self.dim == 5):
         self.__Q = PolynomialRing(QQ, 5, 'xyztw')
      else:
         self.__Q = PolynomialRing(QQ, self.dim, 'x')
      self.__Q.inject_variables()
```

```

        createDir('__EigSps')
        createDir('__ASets')
        createDir('__Relations')
        createDir('__AHilb')
```

All the methods that follow are defined within the SymQuotSing class. To improve efficiency, the class may internally store data for future usage without the need for re-computation. `__str__` creates a string that describes the created object, while `filename_str` creates a string used for storing the computed data.

```
    def __str__(self):
        return "Quotient of CC^"+str(self.dim)+" by the \
                group (ZZ/"+str(self.r)+")^"+str(self.dim-1)

    def filename_str(self):
        return "sym-"+str(self.dim)+'-'+str(self.r)

    def __ZBasis(self):
        basis = []
        c = self.__L.basis()
        for i in range(self.dim):
            basis.append(self.r*c[i])
        return basis
```

The last method above, `__ZBasis`, creates a list of `dim` vectors that are basis of the sublattice $\mathbb{Z}^n \subset L$. All the lattice points are printed out as their $r$-th multiple, to avoid dealing with fraction $\frac{1}{r}$.

The private recursive method `__latptRec` computes a list of the lattice points in

$$L = \mathbb{Z}^n \oplus \frac{1}{r}(1,-1,0,\ldots,0) \oplus \frac{1}{r}(1,0,-1,\ldots,0) \oplus \frac{1}{r}(1,0,0,\ldots,-1)$$

that are contained in the junior simplex and stores the list into an empty list basket. With the public method `LatticePoints` we can return the data stored in `basket` without the need to state all the private arguments of `__latptRec` that is called internally.

```
    def __latptRec(self,current_vect, remaining_n, \
                                          remaining_r, basket):
        if remaining_n == 0 and remaining_r == 0:
            basket.append(current_vect)
            return
        if remaining_n < 0 or remaining_r <0:
            return

        for i in range(remaining_r+1):
            vs = current_vect + [i]
            self.__latptRec(vs, remaining_n - 1, \
                                remaining_r - i,  basket)
        return

    def LatticePts(self):     #, below = false):
        S = []
        self.__latptRec([], self.dim, self.r, S)

        S.sort();
        return S
```

The next method `weight` takes a monomial and returns the index of the eigenspace it belongs to. The argument `vect` can be either a monomial or an array of its exponents in lexicographical order. The eigenspaces $L_{a_1 a_2 a_3 \ldots a_{n-1}}$ of the group action are labelled by the $n-1$ values $a_i \in \{0,1,\ldots,r-1\}$. The function first computes the $(n-1)$-tuple $a_1 a_2 \ldots a_{n_1}$ and in the next step treats it as an integer written in base $r$. The return value is the value of this integer in decimal base.

```
    def weight(self, vect):
        wt = 0
        if hasattr(vect, "exponents"):
            vect = vect.exponents()[0]
        mult = 1
        for i in range(self.dim-1,0,-1):
            wt += ((vect[0] - vect[i])%self.r)*mult
            mult *= self.r
        return wt
```

The following two methods are used to compute the minimal generators of each eigenspace, viewed as a module over the invariant ring. We run through all of the monomials dividing $(x_1 x_2 \ldots x_n)^r$ and put them in eigenspaces they belong to. In `EigSp`, the method checks whether the list of eigenspaces has already been computed, that is if a file "`sym-dim-r__eigsps.p`" exists in the folder `__EigSps`. If yes, the data will just be read and returned. Otherwise, the private method `__eigspRecursion` is called, and the resulting list of lists of generators of the eigenspaces is stored in the previously mentioned file for future use.

```
    def __eigspRecursion(self, ind, currentExponents, EigSp):
        if ind == self.dim -1:
            monomial = 1
            for k in range(self.dim):
                monomial *= self.__Q.gen(k)**currentExponents[k]
            eig = self.weight(monomial)
            survived = true
            for i in range(len(EigSp[eig])):
                if EigSp[eig][i] <> [0]*self.dim and  \
                greaterThan(currentExponents, EigSp[eig][i]):
                survived = false
                break
          if survived:
             EigSp[eig].append(currentExponents)
          return

       for j in range(self.r):
          self.__eigspRecursion(ind+1, \
                               currentExponents + [j], EigSp)

    def EigSp(self, exponents_only=false):
       fileName ='__EigSps/'+self.filename_str()+'__eigsps.p'
       if os.path.exists(filename):
          f = open(fileName, 'rb')
          EigSp = cPickle.load(f)
          f.close()

       else:
          EigSp = []
          for i in range(self.ord):
             EigSp.append([])
          for j in range(self.dim):
             t = [0]*self.dim
             t[j] = self.r
             EigSp[0].append(t)
           self.__eigspRecursion(-1, [], EigSp)
           f = open(fileName, 'wb')
           cPickle.dump(EigSp, f)
           f.close()
       if exponents_only:
          return EigSp
       for i in range(self.ord):
          for j in range(len(EigSp[i])):
             monom = 1
             for k in range(self.dim):
                monom *= self.__Q.gen(k)**EigSp[i][j][k]
             EigSp[i][j] = monom
       return EigSp
```

The key tree-traversal algorithm, producing all the monomial ideals in $\mathbb{C}[x_1, \ldots, x_n]$ that define a $G$-cluster is contained in the method ASets. It is a slight modification of the method of the same name in the Magma code written by Reid [2]. The main idea is that we pick a single monomial from the first non-trivial eigenspace (index 1 in the list of lists returned by EigSp), and add the remaining monomials from this eigenspace to the list of generators of an ideal I. We continue the same prosecc with the remaining eigenspaces one by one and so on. The process either ends with exactly $r^{n-1}$ chosen monomials, one from each eigenspace so the ideal I defines a $G$-cluster, or the ideal becomes "too big" and contains a whole following eigenspace, so it does not describe a $G$-cluster. In both cases, we backtrack one step, and choose a different monomial from the previous eigenspace. As with the method EigSp, the data is stored into a file after being computed for the first time.

```
def ASets(self, exponents_only = false):
    fileName = '__ASets/'+self.filename_str()+'__a-sets.p'
    if os.path.exists():
        f = open(fileName, 'rb')
        asets = cPickle.load(f)
        f.close()
        for i in range(len(asets)):
            for j in range(len(asets[i])):
                monom = 1
                for k in range(self.dim):
                    monom *= self.__Q.gen(k)**asets[i][j][k]
                asets[i][j] = monom
        return asets

    EigSp = self.EigSp()
    I = []
    C = []
    M = []
    asets = []

    finished = false
    while not finished:

        S = exclude(EigSp[0],1)
        over = true
        max_i = -1
        for i in range(len(I)):
            S += exclude(EigSp[I[i]], C[i][M[i]])
            if len(C[i]) != M[i]+1:
                over = false
                max_i = i
        Id = self.__Q.ideal(S)
        Qbar = self.__Q.quotient_ring(Id)
        remaining = []
        exists_empty = false

        for i in range(1, self.ord):
            surviving_i = []
            for j in range(len(EigSp[i])):
                el = EigSp[i][j]
                if el not in Id:
                    quot_el = Qbar.lift(Qbar.retract(el))
                    surviving_i.append(quot_el)
            if len(surviving_i) != 1:
                remaining.append([i,surviving_i])
            if len(surviving_i) == 0:
                exists_empty = true

        if len(remaining) == 0:
            asets.append(Qbar.defining_ideal().\
                                        interreduced_basis())

        if len(remaining) != 0 and exists_empty == false:
            I.append(remaining[0][0])
            C.append(remaining[0][1])
            M.append(0)
        if len(remaining) == 0 or (len(remaining)!=0 and \
                                        exists_empty ==true):
            if over:
```

```
            finished = true
         else:
            broj = len(I) - max_i - 1
            remove_last(I, broj)
            remove_last(M, broj)
            remove_last(C, broj)
            M[max_i] += 1

   asets2 = []
   for i in range(len(asets)):
      asets2.append([])
      for j in range(len(asets[i])):
         asets2[i].append( asets[i][j].exponents()[0])
   f = open(fileName, 'wb')
   cPickle.dump(asets2, f)
   f.close()

   if exponents_only:
      return asets2

  return asets
```

The following method `EigenBases`, based on the corresponding `__eigenbases`, returns a list, each entry of which is a list of exactly `ord` monomials forming a basis for $\mathbb{C}[x_1, \ldots, x_n] / \mathcal{I}_Z$ corresponding to the cluster $Z$ defined by the ideal $\mathcal{I}_Z$ obtained from `ASets`.

```
def __eigenbasis(self, S):
   m = 1
   for i in range(self.dim):
      m *= self.__Q.gen(i)**self.r
   basis = Set( self.__Q.monomial_all_divisors(m) )
    for I in range(len(S)):
      opp = self.__Q.monomial_quotient(m, S[I])
      L = self.__Q.monomial_all_divisors(opp)
      basis -= Set(L)

```

```
        clus = [0]*self.ord

        for i in range(self.ord):
            mono = self.__Q.monomial_quotient(m, basis[i])
            ind = self.weight(mono)
            clus[ind] = mono
        return clus

    def EigenBases(self):
        fileName = '__EigenBases/' + self.filename_str() + \
                                                        \'__eigbas.p'
        if os.path.exists(fileName):
            f = open(fileName, 'rb')
            basket  = cPickle.load(f)
            f.close()

        else:
            basket = []
            A = self.ASets()
            for i in range(len(A)):
                basket.append( self.__eigenbasis(A[i]) )
            f = open(fileName, 'wb')
            cPickle.dump(basket, f)
            f.close()
         return basket
```

Once we obtain the bases of the vector space $\mathcal{O}_Z$, for a $G$-invariant cluster $Z$ (using method `EigenBases`, we can deform the equations in $\mathcal{I}_Z$ and obtain an affine piece parametrising $G$-clusters with the origin being the torus invariant cluster $Z$. For each entry of the list `EigenBases`, method `__relations` returns a list of $G$-invariant ratios of monomials that correspond to the coordinates of the affine piece. For example, when $r = 3$ and dimension $n = 4$, one of the ratios at index 1 looks like `[1,-1,-1,-1]` and this corresponds to the relation $x = \lambda yzt$ for some value of $\lambda \in \mathbb{C}$.

The method `__relations` creates a list of ratios in the following way. An element from the monomial ideal `X.ASets()[i]` is paired with an element from the basis of $\mathcal{O}_Z$ that lies in the same eigenspace. Once this is done, the function calls `__reduce_rels` to obtain a minimal set of relations, by removing the relations that are multiples of other relations from the list. To ensure the minimality, it randomly permutes the entries of the list containing the current relations, and runs the `__reduce_rels` again. Once no changes are made, the process stops. The function `Relations` simply iterates `__relations` over all the monomial ideals of $G$-clusters.

```
def __reduce_rels(self, mat):
            M = matrix(mat)
        K = M.kernel().matrix().rows()
        indices = []

        for i in range(len(K)):
            npos = 0
            nneg = 0
            indp = indn = -1
            for j in range(len(mat)):
                if K[i][j] > 0:
                    npos += 1
                    indp = j
                else:
                    if K[i][j] < 0:
                        nneg += 1
                        indn = j
            if npos == 1  and K[i][indp] ==1 and nneg > 0:
                indices.append(indp)
            else:
                if npos > 0 and nneg == 1 and K[i][indn]==-1:
                    indices.append(indn)
        M = M.delete_rows(indices)
        return M.rows()

def __relations(self, aset, basis):
        mat = []
        for i in range(len(aset)):
            bb = aset[i].exponents()[0]
            ind = self.weight(bb)
            cc = basis[ind].exponents()[0]
            #print bb, cc
            row = []
            for i in range(len(bb)):
                row.append(bb[i] - cc[i])
            mat.append( row )

        redmat = self.__reduce_rels(mat)
        if len(redmat) != self.dim:
            random.shuffle(redmat)
        while (redmat != mat):
            mat = redmat
            redmat = self.__reduce_rels(mat)
        for i in range(self.r):
            random.shuffle(redmat)
            mat = redmat
            redmat = self.__reduce_rels(mat)
        return mat

    def Relations(self):
        fileName = '__Relations/' + self.filename_str() + \
                                                         '__rels.p'
        if os.path.exists(fileName):
            f = open(fileName, 'rb')
            basket  = cPickle.load(f)
            f.close()

        else:
            basket = []
            A = self.ASets()
            B = self.EigenBases()
            for i in range(len(A)):
                basket.append( self.__relations(A[i], B[i]) )
            f = open(fileName, 'wb')
            cPickle.dump(basket, f)
            f.close()
        return basket
```

Finally, once the relations are computed, the private method `__affinepiece` checks whether there are exactly $n$ generators. If this is true, the $n$-dimensional affine piece has exactly $n$ coordinates so it must be a copy of $\mathbb{C}^n$. The method `_affinepiece` then takes the adjoint of the matrix which gives the vertices of the toric cone of the affine piece. As with the prior pair of private and public methods, the method `AHilbFan` iterates `__affinepiece` over all the computed $G$-clusters and returns a list of cones, where a cone is represented by a list of its vertices. In low dimensions, the data obtained from `AHilbFan` can be used to plot the fan, using the in-built Sage function `plot` or `plot3d`.

```
    def __affinepiece(self, mat):
        pts = []
        if len(mat) == self.dim:
            A = (1/self.r**(self.dim-2))*matrix(mat).adjoint()
            for i in range(self.dim):
                if A[0][i] < 0:
                    A *= -1
                    break
                if A[0][i] > 0:
                    break
            pts = A.columns()
        return pts

    def AHilbFan(self):
        fileName = '__AHilb/'+self.filename_str()+'__AHilb.p')
        if os.path.exists():
            f = open(fileName, 'rb')
            pieces  = cPickle.load(f)
            f.close()

        else:
            matrices = self.Relations()
```

```
         pieces = []
         for i in range(len(matrices)):
            pieces.append( self.__affinepiece(matrices[i]) )
         f = open(fileName, 'wb')
         cPickle.dump(pieces, f)
         f.close()
      return pieces
```

In addition to the class methods, there are several external methods we use. The function `createDir` takes a string as an argument and creates a directory with the name specified in the string.

```
def createDir(filename):
        try:
                if not os.path.exists(filename):
                        os.makedirs(filename)
        except OSError:
                print "Error: cannot create the folder"
```

The function `greaterThan` takes two vectors and returns `True` if every entry of the first vector is smaller or equal to the corresponding entry of the first vector. We use it to check whether a monomial divides another monomial in cases where monomials are represented by the list of their exponents.

```
def greaterThan(vector1, vector2):
        n = len(vector1)
        try:
                for i in range(n):
                        if (vector1[i] < vector2[i]):
                                return false
                return true
        except IndexError:
                print("Error: vectors of neq dimensions")
```

The final two functions, `exclude` and `remove_last` deal with lists. The first one `exclude` takes two arguments: a list and a potential element of a list. It returns a copy of the list, but without the element from the argument. Notice that it does not change the original list. The function `remove_last`, however, does change the list it takes as an argument, and simply removes the last `n` elements from it.

```
def exclude(lis, element):
        copyL = []
        for i in range(len(lis)):
                if lis[i] != element:
                        copyL.append(lis[i])
        return copyL

def remove_last(lis, n):
        for i in range(n):
                lis.pop()
```

## 2 Usage

Let a group $G = (\mathbb{Z}/2)^{\oplus 3}$ act on $\mathbb{C}^4$ by (??). To define an object of type `SymQuotSing` corresponding to this quotient variety, one needs to pass the value $r$ and the dimension to the constructor:

```
\begin{minted}{python}
sage: X = SymQuotSing(2,4)
Defining x, y, z, t
sage: print X
Quotient of CC^4 by the group (ZZ/2)^3
```

If one later needs to check the dimension, the exponent $r$ of the group, or its order, type:

```
sage: X.dim
4
sage: X.r
2
sage: X.ord
8
```

The method `LatticePts` lists all the lattice points contained in the junior simplex. The output `[0, 0, 1, 1]` refers to the point $\frac{1}{2}(0, 0, 1, 1)$.

```
sage: X.LatticePts()
[[0, 0, 0, 2], [0, 0, 1, 1], [0, 0, 2, 0], [0, 1, 0, 1],
 [0, 1, 1, 0], [0, 2, 0, 0], [1, 0, 0, 1], [1, 0, 1, 0],
 [1, 1, 0, 0], [2, 0, 0, 0]]
```

The eigenspaces of the group action can be obtained by simply typing `X.EigSp()`. As we can see, the first entry, `X.EigSp()[0]`, consists of the generators of the ring of invariants, while the others are generators of the non-trivial eigenspaces over the ring of invariants.

```
sage: X.EigSp()
[[x^2, y^2, z^2, t^2, 1, x*y*z*t],
 [t, x*y*z],
 [z, x*y*t],
 [z*t, x*y],
 [y, x*z*t],
 [y*t, x*z],
 [y*z, x*t],
 [y*z*t, x]]
```

Running any of the commands `X.ASets()`, `X.EigenBases`, `X.Relations()` or `X.AHilb()` prints the long lists of the data. All of this four lists have the same length, determining the Euler number for the irreducible variety $\mathrm{Hilb}^{\mathrm{G}}(\mathbb{C}^n)$ that this program computes. Below we check how many affine pieces there are for the object `X` and we print the first three monomial ideals.

```
sage: len( X.ASets() )
27
sage: X.ASets()[0:3]
[[y^2, z^2, t^2, x],
 [y*z*t, x^2, x*y, y^2, x*z, z^2, x*t, t^2],
 [x^2, x*y, y^2, x*z, y*z, z^2, t^2]]
```

Finnaly, we show how to list all the data corresponding to a $G$-cluster: the generators of the ideal, the monomial basis, the relations and the vertices of the toric cone of the affine piece. Here we do so for the first two entries:

```
sage: for i in range(0,2):
....:     print i
```

```
....:     print "ideal: ", X.ASets()[i]
....:     print "basic monomials: ", X.EigenBases()[i]
....:     print "relations: ", X.Relations()[i]
....:     print "toric cone: ",  X.AHilb()[i]
....:     print " "
....:
0
ideal:  [y^2, z^2, t^2, x]
basic monomials:  [1, t, z, z*t, y, y*t, y*z, y*z*t]
relations: [(0, 0, 0, 2),
            (0, 2, 0, 0),
            (0, 0, 2, 0),
            (1, -1, -1, -1)]
toric cone: [(1, 0, 0, 1),
             (1, 1, 0, 0),
             (1, 0, 1, 0),
             (2, 0, 0, 0)]

1
ideal:  [y*z*t, x^2, x*y, y^2, x*z, z^2, x*t, t^2]
basic monomials:  [1, t, z, z*t, y, y*t, y*z, x]
relations:  [(1, -1, 1, -1),
             (-1, 1, 1, 1),
             (1, -1, -1, 1),
             (1, 1, -1, -1)]
toric cone:  [(1, 0, 1, 0),
              (1, 1, 1, 1),
              (1, 0, 0, 1),
              (1, 1, 0, 0)]
```

## References

[1] The Sage Developers. **SageMath, the Sage Mathematics Software System (Version 7.4)**, 2016. `http://www.sagemath.org`.

[2] Miles Reid. A-Hilb $\mathbb{A}^4$, some computations, counterexamples and conjectures. preprint on webpage at `http://homepages.warwick.ac.uk/~masda/McKay/AH4.pdf`.