Y0_cohomology_ring

June 18, 2017

1 Type II String Theory on Calabi-Yau Manifolds with Torsion and Non-Abelian Discrete Gauge Symmetries

Companion worksheet for https://arxiv.org/abs/1702.08071

The paper describes a 6-dimensional Calabi-Yau manifold with a non-trivial cup product between two degree-two torsion classes. As described in the paper, the cup product can be detected by a (non-orientable) 4-dimensional submanifold Y_0, see Section 4.2 of the paper. In this worksheet, we construct a Δ-complex for Y_0 and determine the cohomology ring structure.

Note: To run this worksheet yourself you need Sage (http://sagemath.org) and the Python package in the `torus_triangulation` *directory of this repository. The latter is only used to build up the cubical/simplicial/Δ-complex representation of the $Y_0 = T^4/(\mathbb{Z}_2 \times \mathbb{Z}_2)$ quotient. The computation of cohomology groups and the cup product is taken from Sage.*

```
In [3]: from __future__ import print_function, absolute_import
        from sage.all import *
        from torus_triangulation.cube_triangulation import CubeTriangulation
        from torus_triangulation.builder import TorusQuotientBuilder
```

1.1 Constructing Y_0

Let $z_i = x_i + \tau y_i$, $i = 0, \ldots, 2$ be the three complex coordinates on the 6-torus covering Y, see Section 3 of the paper. This worksheet constructs the 4-d submanifold Y_0 defined by the four coordinates

$$(x_0, x_1, x_2, y_0)$$

subject to the identifications: * The torus $x_0 \sim x_0 + 2$, $x_1 \sim x_1 + 2$, $x_2 \sim x_2 + 2$, and $y_0 \sim y_0 + 1$. Note the different domain on y_0, which we chose for convenience. * The $G = \mathbb{Z}_2 \times \mathbb{Z}_2$ group action:

```
In [4]: def f(point):
            return tuple([ point[0] + 1,     -point[1],     -point[2],    point[3]],)

        def g(point):
            return tuple([    -point[0], point[1] + 1, -point[2] + 1, -point[3]])
```

Up to the G-action, we need two unit 4-cubes to cover Y_0. These are defined below, together with a particular order of their vertices. Combined with a particular choice of triangulation of these unit 4-cubes that is defined by the `CubeTriangulation` Python class, and as long as everything is invariant under the torus identification and G-action, this defines a decomposition into simplicies with an order on the vertices, that is, a Δ-complex.

In fact, most orderings for the vertices clash with the (torus and/or G-) identifications and fail to define a Δ-complex. It is a non-trivial fact that there is an ordering that works at all, in general the existence is only guaranteed after subdivision of the simplicies. However, by a computer search we found the following solution:

```
In [5]: order1 = [
            (1, 1, 1, 0),
            (1, 1, 1, 1),
            (0, 1, 1, 0),
            (0, 1, 1, 1),
            (0, 0, 0, 1),
            (0, 0, 0, 0),
            (1, 0, 0, 1),
            (1, 0, 0, 0),
            (0, 0, 1, 1),
            (1, 0, 1, 1),
            (0, 0, 1, 0),
            (1, 1, 0, 0),
            (1, 0, 1, 0),
            (0, 1, 0, 0),
            (1, 1, 0, 1),
            (0, 1, 0, 1),
        ]
        order2 = [(o[0]+1, o[1], o[2], o[3]) for o in order1]
```

Given this data, we now build up all simplices on the covering space in the fundamental region $0 \leq x_0, x_1, x_2 \leq 2, 0 \leq y_0 \leq 1$

```
In [6]: builder = TorusQuotientBuilder(2, 2, 2, 1, group_gens=[f, g])

        cube_triangulation = CubeTriangulation(*order1)
        for simplex in cube_triangulation:
            builder.add_simplex_orbit(*simplex)

        cube_triangulation = CubeTriangulation(*order2)
        for simplex in cube_triangulation:
            builder.add_simplex_orbit(*simplex)

        builder.test()   # Test the construction
```

For example, there are 626 two-simplices, the first 5 of which are defined by the vertices

```
In [7]: list(builder.cells.simplices[3])[0:5]

Out[7]: [((1, 0, 2, 0), (2, 1, 1, 0), (1, 1, 1, 0)),
         ((1, 0, 2, 0), (1, 1, 1, 1), (1, 0, 1, 1)),
```

```
        ((1, 2, 0, 0), (2, 1, 1, 0), (1, 2, 1, 0)),
        ((1, 0, 2, 0), (1, 1, 2, 0), (2, 1, 2, 1)),
        ((0, 0, 2, 1), (0, 0, 2, 0), (1, 0, 1, 1))]
```

1.2 Cohomology groups: The fundamental region

As a toy example, consider the $[0,2] \times [0,2] \times [0,2] \times [0,1]$ fundamental region before identifying any cells:

```
In [8]: cube = builder.cells.delta_complex();  cube
```

```
Out[8]: Delta complex with 54 vertices and 1844 simplices
```

As expected, it has the cohomology groups of a point:

```
In [9]: cube.cohomology(reduced=False)
```

```
Out[9]: {0: Z, 1: 0, 2: 0, 3: 0, 4: 0}
```

1.3 Cohomology Groups: Four-torus

Let us quickly check that we indeed obtain a T^4 by identifying opposing sides (and ignoring the G-action):

```
In [10]: T4 = builder.torus_cells.delta_complex();  T4
```

```
Out[10]: Delta complex with 8 vertices and 1201 simplices
```

```
In [11]: T4.cohomology(reduced=False)
```

```
Out[11]: {0: Z, 1: Z x Z x Z x Z, 2: Z^6, 3: Z x Z x Z x Z, 4: Z}
```

1.4 Cohomology groups of Y_0

Finally, we construct the Δ-complex for the $Y_0 = T^4/G$ quotient

```
In [12]: Y0 = builder.quotient_cells.delta_complex();  Y0
```

```
Out[12]: Delta complex with 2 vertices and 301 simplices
```

```
In [13]: Y0.cohomology(reduced=False)
```

```
Out[13]: {0: Z, 1: 0, 2: Z x C2 x C4 x C4, 3: Z x Z, 4: C2}
```

1.5 Cup Products

By dimension, the only interesting case is $H^2(Y_0,\mathbb{Z}) \times H^2(Y_0,\mathbb{Z}) \to H^4(Y_0,\mathbb{Z})$. We start by extracting generators in degree 2, that is, 2-cochains:

```
In [14]: chains2 = Y0.n_chains(2, base_ring=ZZ, cochains=True)
         h2_0, h2_1, h2_2, h2_3 = Y0.cohomology(generators=True, dim=2)
         c0 = chains2.from_vector(h2_0[1].vector(2))
         c1 = chains2.from_vector(h2_1[1].vector(2))
         c2 = chains2.from_vector(h2_2[1].vector(2))
         c3 = chains2.from_vector(h2_3[1].vector(2))
```

We now verify that the chosen generators of $H^2(T^4/G)$ are: * c0 is 2-torsion * c1, c2 are 4-torsion * c3 is free

```
In [15]: for c in [c0, c1, c2, c3]:
             print(c.is_cocycle(), c.is_coboundary(), (2*c).is_coboundary(), (4*c).

True False True True
True False False True
True False False True
True False False False
```

For the codomain of the cup product we also need to chose a generator, which we take to be the 4-cochain `y0`:

```
In [16]: chains4 = Y0.n_chains(4, base_ring=ZZ, cochains=True)
         h4_0, = Y0.cohomology(generators=True, dim=4)
         y0 = chains4.from_vector(h4_0[1].vector(4))
```

The generator of $H^4(T^4/G)$ is `y0`, and we verify that it is 2-torsion:

```
In [17]: y0.is_cocycle(), y0.is_coboundary(), (2*y0).is_coboundary()

Out[17]: (True, False, True)
```

By checking which degree-4 expressions are coboundaries, we can easily build up a list of all cup products:

```
In [18]: [c0.cup_product(c0).is_coboundary(),
          c0.cup_product(c1).is_coboundary(),
          (c0.cup_product(c2) - y0).is_coboundary(),
          c0.cup_product(c3).is_coboundary(),

          c1.cup_product(c0).is_coboundary(),
          c1.cup_product(c1).is_coboundary(),
          (c1.cup_product(c2) - y0).is_coboundary(),
          (c1.cup_product(c3) - y0).is_coboundary(),
```

```
 (c2.cup_product(c0) - y0).is_coboundary(),
 (c2.cup_product(c1) - y0).is_coboundary(),
 c2.cup_product(c2).is_coboundary(),
 c2.cup_product(c3).is_coboundary(),

 c3.cup_product(c0).is_coboundary(),
 (c3.cup_product(c1) - y0).is_coboundary(),
 c3.cup_product(c2).is_coboundary(),
 c3.cup_product(c3).is_coboundary(),
]
```

Out[18]:
```
[True,
 True,
 True,
 True,
 True,
 True,
 True,
 True,
 True,
 True,
 True,
 True,
 True,
 True,
 True,
 True]
```

In [19]:
```
def trivial_cup_product_table(*cohomology_generators):
    """
    Make a table whose entries are whether the cup product is trivial
    """
    names = ['c0', 'c1', 'c2', 'c3']
    rows = [[''] + names]
    for row_name, c in zip(names, cohomology_generators):
        row = [row_name]
        for d in cohomology_generators:
            cd = c.cup_product(d)
            assert cd.is_cocycle()
            row.append('0' if cd.is_coboundary() else '1')
        rows.append(row)
    return table(rows)
```

In [20]:
```
trivial_cup_product_table(c0, c1, c2, c3)
```

Out[20]:

	c0	c1	c2	c3
c0	0	0	1	0
c1	0	0	1	1
c2	1	1	0	0
c3	0	1	0	0

1.6 Alternate Basis

There is a slightly better basis choice that leads to fewer non-trivial table entries:

```
In [21]: trivial_cup_product_table(c0, c2, c1+c0, c3+2*c2)
```

Out[21]:

	c0	c1	c2	c3
c0	0	1	0	0
c1	1	0	0	0
c2	0	0	0	1
c3	0	0	1	0