COMPUTING $\widehat{HF}$ USING THE BORDERED FLOER PACKAGE

ROBERT LIPSHITZ, PETER S. OZSVÁTH, AND DYLAN P. THURSTON

Abstract. How to use the bordered Sage package to compute $\widehat{HF}$.

1. A brief introduction

We start with three computations; we'll explain what's going on briefly here, and in more detail in later sections. *However:* if you've never used Sage before, you'll probably be lost, so take a look at the Sage tutorial first.

As a first, very high-level example, let's compute $\widehat{HF}$ of the branched double cover of the $(-2,3,5)$ pretzel knot.

```
sage: load ChainCx.sage
sage: load PMCAlg.sage
sage: load TypeDStr.sage
sage: load TypeDDStr.sage
sage: load Plats.sage
sage: my_grp = BraidGrp(6)
sage: my_grp.hf_of_knot(5*[1]+3*[3]+2*[-5])
1
```

This takes a while (15 minutes or so, on a modern laptop); you can instead use my_grp.hf_of_knot(5*[1]+3*[3]+2*[−1], True) to have it print how far along it is. As a reminder, in Python, 5*[1]+3*[3]+2*[-5] stands for the list [1,1,1,1,1,3,3,3,-5,-5].

The argument to BraidGrp is the number of strands. The argument for hf_of_knot is the braid word; this one is $\sigma_1^5\sigma_3^3\sigma_5^{-2}$. It will be faster if one uses as few σ_{n-1}'s as possible (if n is the number of strands). Really, this is taking the plat closure of the braid, where σ_2 and σ_4 act trivially on the plat.

If you're doing several computations this way, use the same instance of BraidGrp for all of them: it remembers part of what it has already computed, to speed up repeated computations.

Next, let's compute, in a more hands-on way, the invariant of some lens spaces:

```
sage: load ChainCx.sage
sage: load PMCAlg.sage
sage: load TypeDStr.sage
sage: load TypeDDStr.sage
sage: torus = split_matching(1)
sage: hb0 = zero_type_D(1)
```

RL was supported by NSF Grant DMS-0905796 and a Sloan Research Fellowship.
PSO was supported by NSF grants number DMS-0505811 and FRG-0244663.
DPT was supported by a Sloan Research Fellowship.

```
sage: s1s2 = hb0.mor_to_d(hb0)
sage: s1s2
Chain complex with 2 generators.
sage: s1s2.homology()
2
sage: taul = Arcslide(torus,2,1)
sage: hb1 = taul.dd_mod().mor_to_d(hb0)
sage: hb2 = taul.dd_mod().mor_to_d(hb1)
sage: hb3 = taul.dd_mod().mor_to_d(hb2)
sage: len(hb3.basis)
154
sage: hb3.simplify()
sage: len(hb3.basis)
4
sage: l31 = hb3.mor_to_d(hb0)
sage: l31
Chain complex with 5 generators.
sage: l31.homology()
3
sage: l31
Chain complex with 3 generators.
sage: taum = Arcslide(torus, 1, 0)
sage: hb4 = taum.dd_mod().mor_to_d(hb3)
sage: l32 = hb4.mor_to_d(hb0)
sage: l32.homology()
3
```

We start by loading the relevant code. The command split_matching(1) generates the split pointed matched circle of genus 1. zero_type_D(1) generates $\widehat{CFD}$ of the 0-framed handlebody of genus 1 (i.e., a particular solid torus). Arcslide(torus,2,1) is the arc-slide given by sliding point 2 down over point 1 in this pointed matched circle. (Points are numbered from bottom to top, starting at 0.)

hb0.mor_to_d(hb0) generates the chain complex of morphisms from the hb0 to itself. We then take its homology. (Not surprisingly, it doesn't change.)

The code taul.dd_mod().mor_to_d(hb0) generates the type DD module for this arc-slide, and then the chain complex of morphisms from it to hb0.

len(hb3.basis) gives the number of generators of hb3 as a type D structure. The Hom pairing theorem tends to create lots of extra generators. We cancel these (to speed up computation) with hb3.simplify(). (The names of the generators also get more and more complicated; it's worth occasionally running hb3.shorten_names() to truncate them again.)

Note that taking homology of a chain complex is a destructive operation, as we see with l31.

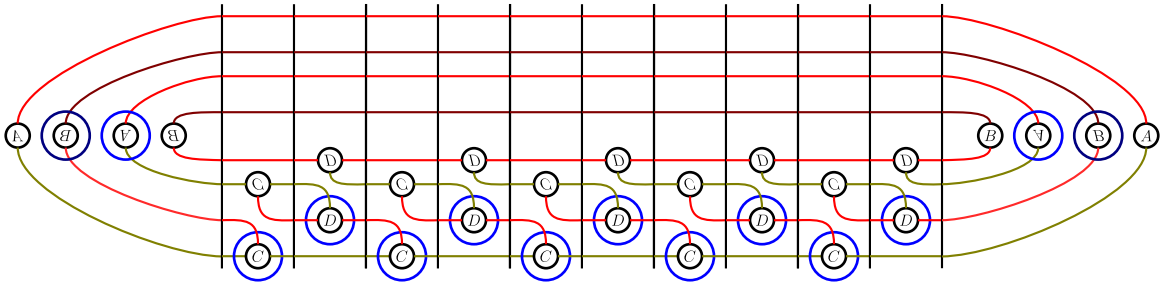
As a more interesting example, we compute $\widehat{HF}$ of the Poincaré homology sphere, and $\Sigma(2,3,7)$.

```
sage: hb = dict()
```

```
sage: hb[0]=zero_type_D(2)
sage: slideseq= [(4,3),(1,0),(2,1),(3,2),(1,0),(5,4),(6,5),(1,2)]
sage: ddslideseq = dict()
sage: for i in range(len(slideseq)):
          ddslideseq[i]=Arcslide(hb[i].pmc, slideseq[i][0],
slideseq[i][1])
          hb[i+1]=ddslideseq[i].dd_mod().mor_to_d(hb[i])
          print "Before simplification, hb["+repr(i+1)+"] has
rank "+repr(len(hb[i+1].basis))
          hb[i+1].simplify()
          print "After simplification, hb["+repr(i+1)+"] has
rank "+repr(len(hb[i+1].basis))
          hb[i+1].shorten_names()
Before simplification, hb[1] has rank 34
After simplification, hb[1] has rank 2
Before simplification, hb[2] has rank 56
After simplification, hb[2] has rank 2
Before simplification, hb[3] has rank 57
After simplification, hb[3] has rank 1
Before simplification, hb[4] has rank 31
After simplification, hb[4] has rank 1
Before simplification, hb[5] has rank 33
After simplification, hb[5] has rank 1
Before simplification, hb[6] has rank 50
After simplification, hb[6] has rank 2
Before simplification, hb[7] has rank 130
After simplification, hb[7] has rank 2
Before simplification, hb[8] has rank 134
After simplification, hb[8] has rank 4
sage: desired_hb = hb[8]
sage: dehna = Arcslide(split_matching(2),1,0)
sage: dehnb = Arcslide(split_matching(2),2,1)
sage: twistitup = dict()
sage: twistitup[0]=desired_hb
sage: for i in range(7):
          tempmod = dehna.dd_mod().mor_to_d(twistitup[i])
          print "Before simplification, b(ab)^"+repr(i)+
"*HB has rank "+repr(len(tempmod.basis))
          tempmod.simplify()
          print "After simplification, b(ab)^"+repr(i)+
"*HB has rank "+repr(len(tempmod.basis))
          twistitup[i+1]=dehnb.dd_mod().mor_to_d(tempmod)
          print "Before simplification, (ab)^"+repr(i+1)+
"*HB has rank "+repr(len(twistitup[i+1].basis))
          twistitup[i+1].simplify()
          print "After simplification, (ab)^"+repr(i+1)+
```

```
"*HB has rank "+repr(len(twistitup[i+1].basis))
          twistitup[i+1].shorten_names()
Before simplification, b(ab)^0*HB has rank 229
After simplification, b(ab)^0*HB has rank 7
Before simplification, (ab)^1*HB has rank 317
After simplification, (ab)^1*HB has rank 5
Before simplification, b(ab)^1*HB has rank 250
After simplification, b(ab)^1*HB has rank 6
Before simplification, (ab)^2*HB has rank 263
After simplification, (ab)^2*HB has rank 7
Before simplification, b(ab)^2*HB has rank 337
After simplification, b(ab)^2*HB has rank 9
Before simplification, (ab)^3*HB has rank 374
After simplification, (ab)^3*HB has rank 10
Before simplification, b(ab)^3*HB has rank 445
After simplification, b(ab)^3*HB has rank 11
Before simplification, (ab)^4*HB has rank 447
After simplification, (ab)^4*HB has rank 13
Before simplification, b(ab)^4*HB has rank 586
After simplification, b(ab)^4*HB has rank 14
Before simplification, (ab)^5*HB has rank 567
After simplification, (ab)^5*HB has rank 15
Before simplification, b(ab)^5*HB has rank 673
After simplification, b(ab)^5*HB has rank 17
Before simplification, (ab)^6*HB has rank 678
After simplification, (ab)^6*HB has rank 20
Before simplification, b(ab)^6*HB has rank 869
After simplification, b(ab)^6*HB has rank 19
Before simplification, (ab)^7*HB has rank 751
After simplification, (ab)^7*HB has rank 21
Sage: poincarehs = twistitup[5].mor_to_d(desired_hb)
sage: poincarehs
Chain complex with 405 generators.
sage: poincarehs.homology()
1
sage: sigma237 = twistitup[7].mor_to_d(desired_hb)
sage: sigma237
Chain complex with 551 generators.
sage: sigma237.homology()
3
```

The computation takes a little while, and the print statements are there to keep me from losing hope. What's going on is that program is computing the $\widehat{CFDD}$ for the following diagram:



This is the first long computation, and the result is called desired_hb. It glues two of these together with Dehn twists $(ab)^5$ or $(ab)^7$ in between (where a and b are longitude and meridian for the bottom torus inside the genus 2 surface); this is the second long computation.

2. Pointed matched circles

2.1. **Creating a pointed matched circle.** Pointed matched circles are specified by the PMC class. To declare a pointed matched circle, the inputs are the genus and the matching. The matching is a list of pairs of elements of $\{0, \dots, 4g-1\}$. For example, the following generates the genus 2 split matching and calls it my_pmc:

```
sage: my_pmc = PMC(2,[(0,2),(1,3),(4,6),(5,7)])
sage: my_pmc
Genus 2 pointed matched circle with matching ((0,2), (1,3),
(4,6), (5,7))
```

There are also two pre-defined methods for creating particular kinds of pointed matched circles: split_pmc(g) returns the split pointed matched circle of genus g, and antipodal_pmc(g) returns the antipodal pointed matched circle of genus g.

2.2. **Operations on pointed matched circles.** Every pointed matched circle has a reverse:

```
sage: pmc1 = PMC(3, [(0,6),(1,8),(2,4),(3,5),(7,10),(9,11)])
sage: pmc1.opposite()
Genus 3 pointed matched circle with matching ((0,2), (1,4),
(3,10), (5,11), (6,8), (7,9))
```

This is useful since we define type *DD* bimodules as left-left bimodules; see Section 5.

A more interesting operation is the arc-slide. Given a pointed matched circle my_pmc, my_pmc.arcslide(i,j) returns the pointed matched circle gotten by sliding the foot i over the foot j. For example:

```
sage: my_pmc = split_matching(2)
sage: my_pmc.arcslide(3,4)
Genus 2 pointed matched circle with matching ((0,2), (1,6),
(3,5), (4,7)
```

Of course, i and j must be adjacent integers.

The methods PMC.is_underslide(i,j) and PMC.is_overslide(i,j) return True if sliding i over j is an underslide or overslide, respectively.

2.3. **Other PMC Methods.**

- PMC's can be displayed graphically with PMC.show().
- PMC.selftest checks if PMC is a valid pointed matched circle—though currently, it doesn't actually check if surgery on the pairs of points gives a connected circle.
- PMC.which_pair and PMC.matched_point deal with the matching.
- PMC.idempotents, PMC.alg_basis and PMC.alg_trunc_basis return the idempotents of the algebra A(PMC), a basis of Strand_Diagram's for the algebra, and a basis for the truncated (no multiplicities greater than 1) algebra, respectively.
- PMC.complementary_idem turns an idempotent over PMC into the complementary idempotent over its orientation reverse.
- PMC.zero returns the zero element of the algebra over PMC.
- PMC.chords returns the chords in PMC.

3. The algebra associated to a pointed matched circle

Two different classes are involved in algebra elements. The class Strand_Diagram represents a basis element for the algebra. The class AlgElt holds a sum of Strand_Diagram's. Most of the heavy lifting is done by Strand_Diagram; however, the behavior of Strand_Diagram's under operations is a little inconsistent, so it is usually better to use AlgElt's.

3.1. **Strand_Diagram's.**

3.1.1. *Defining Strand_Diagram's.* The input to a Strand_Diagram is (pmc, strands, left_idem, rig
For example, a full definition of a Strand_Diagram looks like this:

```
sage: my_pmc = split_matching(2)
sage: my_strand_diag = Strand_Diagram(my_pmc,
[(2,5)],[(0,2),(1,3)], [(1,3),(5,7)], 'rho25')
```

The argument $[(2,5)]$ is the list of strands in the Strand_Diagram (in this case, a single strand from point 2 to point 5). $[(0,2),(1,3)]$ is the left idempotent and $[(1,3),(5,7)]$ the right idempotent. The argument 'rho25' is the (optional) name for this element; we'll come back to that.

It is not necessary to give both left and right idempotents; the following three commands give the same element:

```
sage: my_pmc = split_matching(2)
sage: my_strand_diag = Strand_Diagram(my_pmc, [(2,5)],
[(0,2),(1,3)])
sage: my_strand_diag = Strand_Diagram(my_pmc, [(2,5)],
left_idem=[(0,2),(1,3)])
sage: my_strand_diag = Strand_Diagram(my_pmc, [(2,5)],
right_idem=[(1,3),(5,7)])
```

(If only one idempotent is given, it's assumed to be the left one.)

The argument name is a little strange. Its only effect is on what happens when you print the Strand_Diagram: if it has a name, only the name is printed; otherwise, all of the data of the diagram is shown:

```
sage: my_pmc = split_matching(2)
sage: my_strand_diag = Strand_Diagram(my_pmc, [(2,5)],
[(0,2),(1,3)], name='rho25')
sage: same_strand_diag = Strand_Diagram(my_pmc, [(2,5)],
[(0,2),(1,3)])
sage: my_strand_diag
rho25
sage: same_strand_diag
 | LI:[(0, 2), (1, 3)] S:[(2, 5)] RI:[(1, 3), (5, 7)] |
sage: my_strand_diag == same_strand_diag
 True
```

As the example illustrates, naming is (currently) not very robust: two equivalent (and, indeed, equal) Strand_Diagram's can have different names (or one can be named and the other nameless).

3.1.2. *Operations on Strand_Diagram's.* Strand_Diagram's can be multiplied:

```
sage: rho02=Strand_Diagram(my_pmc, [(0,2)],[(0,2),(5,7)])
sage: rho23=Strand_Diagram(my_pmc, [(2,3)],[(0,2),(5,7)])
sage: rho02*rho23
 | LI:[(0, 2), (5, 7)] S:[(0, 3)] RI:[(1, 3), (5, 7)] |
```

They can also be differentiated:

```
sage: rho1523=Strand_Diagram(my_pmc, [(1,5),(2,3)], [(1,3),(0,2)])
sage: rho1523.differential()
[ | LI:[(0, 2), (1, 3)] S:[(1, 3), (2, 5)] RI:[(1, 3), (5, 7)] | ]
```

Warning! If an operation on a Strand_Diagram gives 0, the returned answer is [] (the empty list):

```
sage: rho23*rho02
[]
sage: rho23.differential()
[]
```

This is *not* a Strand_Diagram. This means that, when programing with Strand_Diagram's, one needs to test if operations give 0 or not. Therefore, it's generally safer to work with AlgElt's.

The product can also be called with Strand_Diagram.r_multiply or Strand_Diagram.r_multiply to right or left multiply this element by the argument, respectively.

3.1.3. *Other Strand_Diagram methods.*

- Strand_Diagram's can be displayed graphically with Strand_Diagram.show().
- Strand_Diagram.name_me(name) allows you to name a Strand_Diagram after it has been created.
- Strand_Diagram.multiplicities() returns a list of the local multiplicities of the Strand_Diagram. Strand_Diagram.is_in_trunc() returns True if none of the multiplicities are bigger than 1.
- Gradings: Strand_Diagram.inv() returns the number of inversions in Strand_Diagram; Strand_Diagram.iota() returns $\iota()$; Strand_Diagram.big_grading() returns the G'-grading of a.
- Strand_Diagram.augmentation() takes the augmentation of this diagram, i.e., returns self if it's an idempotent and [] otherwise. Strand_Diagram.id_minus_aug() returns the identity minus the augmentation.
- Strand_Diagram.opposite() returns the "same" strand diagram, but viewed as an element of the opposite algebra.

3.2. **AlgElt's.**

3.2.1. *Defining AlgElt's.* Basically, an AlgElt is a list of Strand_Diagram's ($\mathbb{F}_2$-linear combinations are lists).

```
sage: a = AlgElt([rho02,rho23,rho1523])
```

You can also explicitly specify the pointed matched circle; the following example is equivalent to the previous one:

```
sage: a = AlgElt([rho02,rho23,rho1523],my_pmc)
```

In either case, you can get the pointed matched circle via:

```
sage: a.pmc
Genus 2 pointed matched circle with matching ((0, 2), (1, 3), (4, 6), (5
```

If the list is empty, you *must* specify the pointed matched circle (though currently the program doesn't raise an exception if you don't).

```
sage: zero = AlgElt([],my_pmc)
sage: zero
<[]>
sage: zero.pmc
Genus 2 pointed matched circle with matching ((0, 2), (1, 3), (4, 6), (5
```

3.2.2. *Operations on AlgElt's.* Like Strand_Diagram's, AlgElt's can be multiplied and differentiated:

```
sage: a*a
<[ | LI:[(0, 2), (5, 7)] S:[(0, 3)] RI:[(1, 3), (5, 7)] | ]>
sage: a.differential()
<[ | LI:[(0, 2), (1, 3)] S:[(1, 3), (2, 5)] RI:[(1, 3), (5, 7)] | ]>
```

AlgElt.diff() is the same as AlgElt.differential().

They can also be added:

```
sage: a+a
<[]>
sage: a+AlgElt([rho23,rho02])==AlgElt([rho1523])
True
sage: a+a==0
True
```

Again, unlike Strand_Diagram's, the zero AlgElt *is* an AlgElt. It evaluates as equal to anything else that Python considers False (including the empty list, 0, False and None).

3.2.3. *Other AlgElt methods.* The AlgElt class extends UserDict, so you can get elements from it, iterate over it, and so on:

```
sage: for x in a:
          print x.differential()
[]
[]
[ | LI:[(0, 2), (1, 3)] S:[(1, 3), (2, 5)] RI:[(1, 3), (5, 7)] | ]
```

3.3. **Other algebra operations.**

- AlgElt.opposite() returns the same strand diagrams, but viewed as an element of the algebra over the orientation-reversed surface.
- alg_element(pmc, strands, spinc=0) returns the AlgElt which is a sum of all ways of choosing idempotents compatible with strands. l_idem_compat(pmc,strands,idem) checks if idem is compatible, as a left idempotent, with strands. There is a similar method r_idem_compat testing if a right idempotent is compatible with strands.
- alg_homology(pmc, spinc=0) returns a basis for the homology of the algebra associated to pmc, in the spin^c-structure spinc.
- mul_list(list1, list2) multiplies a list of strand diagrams; better to use the AlgElt class. Similarly, alg_differentiate_list differentiates a list of strand diagrams, aug_list applies the augmentation to a list, and id_minus_aug_list applies id minus the augmentation.
- BigGradingGroup is a class for dealing with G' gradings.
- AlgBlgElt is a class for elements of $\mathcal{A}(\mathcal{Z})\otimes\mathcal{A}(\mathcal{Z}')$, and is used for DD modules.

4. Type D modules

This section has not been written yet.

4.1. **Defining type D modules.** The package knows about several different handlebodies:

- infty_type_D(k)
- zero_type_D(k)
- m_one_type_D(k)

4.2. **Operations on type D modules.**

5. Type DD modules

This section has not been written yet.

5.1. **Defining type DD modules.**

5.2. **Operations on type DD modules.**

Department of Mathematics, Columbia University, New York, NY 10027
E-mail address: lipshitz@math.columbia.edu

Department of Mathematics, Columbia University, New York, NY 10027
E-mail address: petero@math.columbia.edu

Department of Mathematics, Barnard College, Columbia University, New York, NY 10027
E-mail address: dthurston@barnard.edu