String diagrams for higher mathematics with `wiggle.py`

Simon Burton

Quantinuum

February 14, 2023

Abstract

We introduce `wiggle.py` which is a python based library for vector graphics rendering of 3-dimensional string (surface) diagrams for monoidal bicategories. The library uses linear constraint optimization, resulting in a highly customizable layout engine.

1 Introduction

The goal of `wiggle.py` is to render 3-dimensional string diagrams (surface diagrams) for monoidal bicategories, subject to the following requirements:
(1) easy to use "pythonic" interface,
(2) separate mechanism over policy [Ray03], and
(3) include LaTeX text, with pdf/svg/png output.

Like a web framework, `wiggle.py` can do fancy diagrams with one line of code, but things get much more tricky and verbose when customizing. The implementation strategy used was to keep rewriting and refactoring the codebase until it started to satisfy the above requirements. It seems quite impossible to design such a codebase upfront, unless one has previously written such a thing. The resulting design appears somewhat baroque, but for good reasons. `wiggle.py` is more like a compiler than a web framework under the hood.

The layout algorithm is based on linear programming. The user constructs cube-shaped cells (or "bricks"[HH19]) by composing horizontally, vertically and depth-wise. This composition results in a set of weak constraints that may be violated, and strong constraints that cannot be violated. The weak constraints produce an objective function that is minimized; these come from the position of the cells. The strong constraints are equations to solve, which come from gluing surfaces, strings and vertices ("pips") together. The position of the cells serve as a scaffolding within which the surfaces, strings, and vertices reside. Separating the constraints this way allows for customizing the layout: the weak constraints may be pushed around by the user, but there's no reason to allow the strong constraints to be altered.

String diagrams (2-dimensional) for monoidal categories are introduced in [JS91, BS09, Sel10], and are connected to the earlier work [Pen71]. The use of string (and surface) diagrams for (bi-)categories is nicely introduced in [Mar14], and for higher categories in [Hum12, BMS12, Reu17, HV19]. Other python tools include `rewalt` [HK22] which takes a disciplined approach to higher-dimensional structures, while only rendering 2-dimensional string diagrams. The online tool `homotopy.io` [RV19] can render 3-dimensional string diagrams, but has limited customization support, and is javascript based.

The use of string (surface) diagrams in higher category theory is now widespread, for example, see [PS12] Fig. 15. Another approach is to use "movies" or 2-dimensional slices through higher-dimensional diagrams, for example [Car11, Ver17, CP22].

In the appendix below we give an example interactive session using `wiggle.py` .

References

[BMS12] John W Barrett, Catherine Meusburger, and Gregor Schaumann. Gray categories with duals and their diagrams. *arXiv preprint arXiv:1211.0529*, 2012.

[BS09] J Baez and M Stay. Physics, topology, logic and computation: A rosetta stone, in "new structures for physics", 95–172. *Lecture Notes in Phys*, 813, 2009.

[Car11] J Scott Carter. *Excursion In Diagrammatic Algebra: Turning a sphere from red to blue*, volume 48. World Scientific, 2011.

[CP22] Jonathan A Campbell and Kate Ponto. Riemann-Roch theorems in monoidal 2-categories. *arXiv preprint arXiv:2203.04351*, 2022.

[HH19] Jules Hedges and Jelle Herold. Foundations of brick diagrams. *arXiv preprint arXiv:1908.10660*, 2019.

[HK22] Amar Hadzihasanovic and Diana Kessler. Data structures for topologically sound higher-dimensional diagram rewriting. *arXiv preprint arXiv:2209.09509*, 2022.

[Hum12] Benjamin Taylor Hummon. *Surface diagrams for gray-categories.* PhD thesis, UC San Diego, 2012.

[HV19] Chris Heunen and Jamie Vicary. *Categories for Quantum Theory: An Introduction.* Oxford University Press, USA, 2019.

[JS91] André Joyal and Ross Street. The geometry of tensor calculus, i. *Advances in mathematics*, 88(1):55–112, 1991.

[Mar14] Daniel Marsden. Category theory using string diagrams. *arXiv preprint arXiv:1401.7220*, 2014.

[Pen71] Roger Penrose. Applications of negative dimensional tensors. *Combinatorial mathematics and its applications*, 1:221–244, 1971.

[PS12] Kate Ponto and Michael Shulman. Duality and traces for indexed monoidal categories. *Theory and Applications of Categories*, 26(23):582–659, 2012.

[Ray03] Eric S Raymond. *The art of Unix programming.* Addison-Wesley Professional, 2003.

[Reu17] David Reutter. Frobenius algebras, Hopf algebras and 3-categories. *Talk given at Perimeter Institute*, 2017.

[RV19] David J Reutter and Jamie Vicary. Shaded tangles for the design and verification of quantum circuits. *Proceedings of the Royal Society A*, 475(2224):20180338, 2019.

[Sel10] Peter Selinger. A survey of graphical languages for monoidal categories. In *New structures for physics*, pages 289–355. Springer, 2010.

[Ver17] Dominic Verdon. Coherence for braided and symmetric pseudomonoids. *arXiv preprint arXiv:1705.09354*, 2017.

Appendix: example `wiggle.py` session

November 7, 2022

1 Install

The `wiggle` module lives inside the `huygens` package. To install:

```
$ pip install git+https://github.com/punkdit/huygens
```

This has only been tested on linux.

The constraint satisfier tends to complain a lot, so we turn off these warnings.

[51]:
```
import warnings
warnings.filterwarnings('ignore')
```

2 Basic examples

First we make some transparent colors.

[2]:
```
from huygens.namespace import *

pink = color.rgba(1.0, 0.37, 0.36, 0.5)
grey = color.rgba(0.85, 0.85, 0.85, 0.5)
cream = color.rgba(1.0, 1.0, 0.92, 0.5)
cyan = color.rgba(0.0, 0.81, 0.80, 0.5)
yellow = color.rgba(1.0, 0.93, 0.4, 0.5)
```

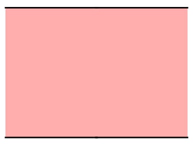
Objects, 1-morphisms and 2-morphisms are represented as the classes `Cell0`, `Cell1` and `Cell2` respectively. We call these 0-cells, 1-cells and 2-cells.

[3]:
```
from huygens.wiggle import Cell0, Cell1, Cell2
```

0-cells are rendered as 2-dimensional regions.

[4]:
```
n = Cell0("n", fill=pink)
n
```

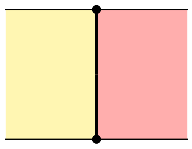
[4]:



A 1-cell is rendered as a 1-dimensional separating boundary between two 0-cells.

```
[5]: m = Cell0("m", fill=yellow)
     A = Cell1(m, n)
     A
```

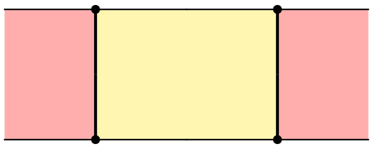
[5]:



For horizontal composition of 1-cells we use the python `<<` operator.

```
[6]: B = Cell1(n, m)
     B<<A
```

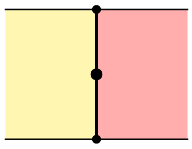
[6]:



2-cells are rendered as a 0-dimensional separating boundary between two 1-cells.

```
[7]: f = Cell2(A, A)
     f
```

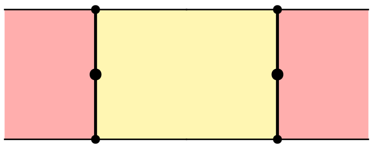
[7]:



These have both a horizontal composition, using `<<` and a vertical composition using the `*` operator.

```
[8]: g = Cell2(B, B)
     g<<f
```

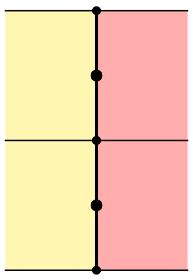
[8]:



[9]:

```
f*f
```

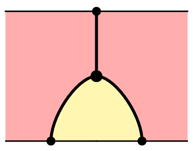
[9]:



[10]:

```
C = Cell1(n, n)
Cell2(C, B<<A)
```

[10]:

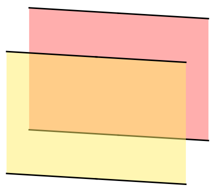


The monoidal product is rendered in the third dimension, front to back. By default, `wiggle.py` switches to an oblique view whenever we have more than one layer.

[11]:

```
m @ n
```

[11]:

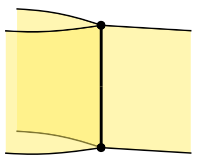


Here we show a (co-)multiplication for a pseudomonoid.

[12]:

```
Cell1(m @ m, m)
```

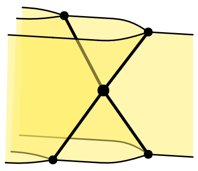
[12]:



The (co-)associativity 2-cell is a bit tricky because we need identity 1-cells, and preferably these should be invisible. We do this by setting the `pip_color` and `stroke` to `None`.

[13]:
```
m_m = Cell1(m, m, pip_color=None, stroke=None)
mm_m = Cell1(m @ m, m)
top = (m_m @ mm_m) << (mm_m)
bot = (mm_m @ m_m) << (mm_m)
assoc = Cell2(top, bot, cone=1.)
assoc
```

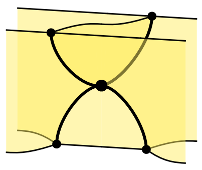
[13]:



The `cone=1.` assignment makes the wires coming out of the 2-cell straight, like a cone. By default, these are more curvy. We show this with a Frobeniator 2-cell.

[14]:
```
m_mm = Cell1(m, m @ m)
src = mm_m << m_mm
tgt = (m_mm @ m_m) << (m_m @ mm_m)
Cell2(tgt, src)
```

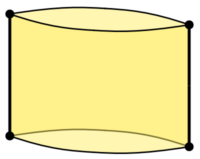
[14]:



When we construct a tube, we need an invisible 0-cell to act as the monoidal identity. We call this `i`:

[15]:
```
i = Cell0("i", stroke=None)
tube = Cell1(i, m @ m) << Cell1(m @ m, i)
tube
```

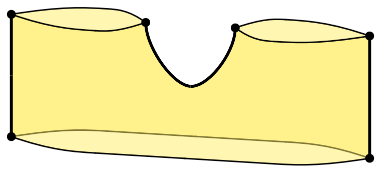
[15]:



Here we construct a saddle and use it to build a pair-of-pants.

[16]:
```
saddle = Cell2(
    Cell1(m @ m, i) << Cell1(i, m @ m),
    m_m @ m_m,
    pip_color=None,
)
lfold = Cell2(Cell1(i, m @ m), Cell1(i, m @ m), pip_color=None)
rfold = Cell2(Cell1(m @ m, i), Cell1(m @ m, i), pip_color=None)
pop = lfold << saddle << rfold
pop
```

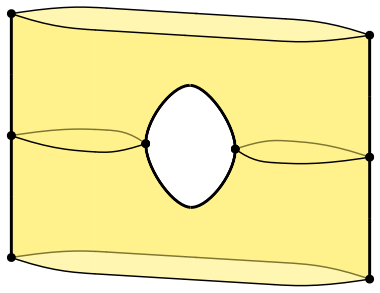
[16]:



We can take horizontal, vertical, or depth-wise reversal of a cell, by using the methods `h_rev`, `v_rev` or `d_rev`.

[17]:
```
pop.v_rev() * pop
```

[17]:

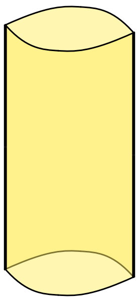


3 Cell layout and render

Once we have finished constructing a cell and are ready for rendering, the next thing that happens is the layout in 3-dimensions. This is where the linear programming constraint solver is called. The `layout` method on cell objects has arguments for controlling the overall dimensions.

[19]:
```
cell = tube.layout(width=1, height=1, depth=1)
cell
```

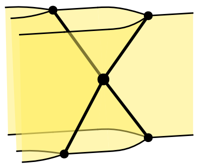
[19]:



After layout comes the render, which happens in the method `render_cvs`. This has argument `pos` which changes the point-of-view.

```
[20]: assoc.render_cvs(pos="northwest")
```

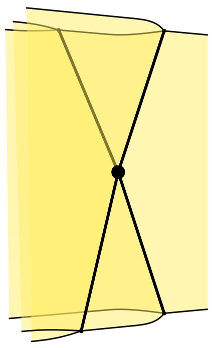
[20]:



The default is orthographic projection, but we can also do non-orthographic projection for a more cinematic effect.

```
[21]: # TODO: pip_radius default is too small for projective render
      cell = assoc(pip_radius=0.2).layout(height=1)

      cell.render_cvs(ortho=False, eyepos=[-0.8, -2., 0])
```

[21]:

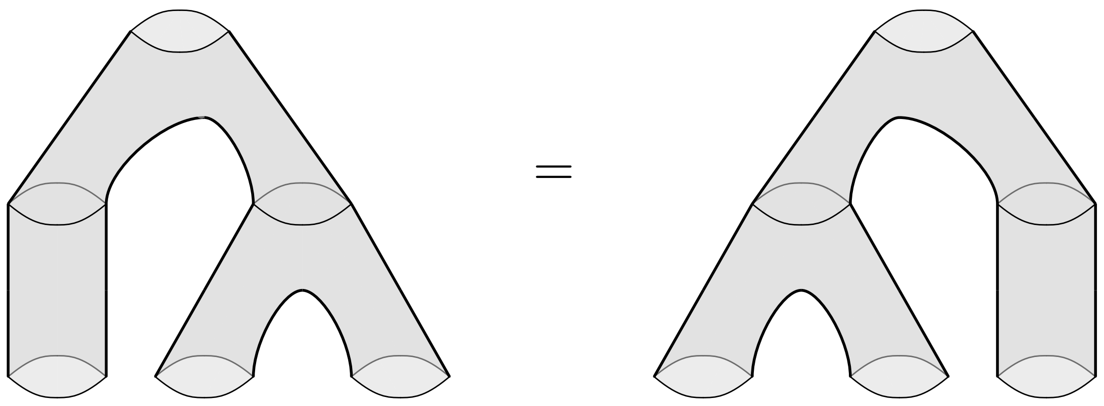


4 More constraints

The pair-of-pants above can be further customized by adding more linear constraints. An example of this is in the helper functions `make_pants`, `make_tube`, and `make_pants_rev`.

[50]:
```
from huygens.wiggle import make_pants, make_tube

m = Cell0("m", fill=grey)
pop = make_pants(m)
tube = make_tube(m)
lhs = pop * (tube << pop)
rhs = pop * (pop << tube)
lhs = lhs.layout(width=2, height=1.5, depth=1).render_cvs(pos="north")
rhs = rhs.layout(width=2, height=1.5, depth=1).render_cvs(pos="north")

cvs = Canvas().insert(0, 0, lhs).insert(10, 0, rhs)
cvs.text(6.5, 2, "$=$", [Scale(2.0)])
```

[50]:



5 Decorating the vertices

We can put labels in the vertex, or an arbitrary decoration. This is done using a `Canvas` object. The cell vertex is called the "pip".

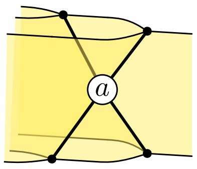
[22]:
```
p = path.circle(0, 0, 0.2)
cvs = Canvas().fill(p, [white]).stroke(p).text(0, 0, "$a$", st_center)
cvs
```

[22]:

We also can override cell's attribute (producing a copy of the object) using the call syntax.

[23]:
```
assoc(pip_cvs=cvs)
```

[23]:

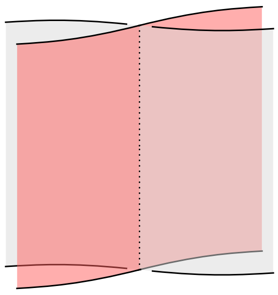


6 More examples

We use the `st_braid` stroke style to show a braid of objects in a monoidal bicategory.

[24]:
```
from huygens.wiggle import st_braid

pos = "northwest"
eyepos=[0.6,-3,1]

l = Cell0("l", fill=pink)
m = Cell0("m", fill=grey)

m_ = m(st_stroke=st_braid)
l_ = l(st_stroke=st_braid)

l_l = Cell1(l, l, pip_color=None, stroke=None)
m_m = Cell1(m, m, pip_color=None, stroke=None)

Rneg = Cell1( l@m_, m_@l, pip_color=None, st_stroke=st_dotted)
Rpos = Cell1( m_@l, l@m_, pip_color=None, st_stroke=st_dotted)

b_neg = Cell2(Rneg, Rneg, pip_color=None)
b_neg.layout(width=1, height=1).render_cvs(pos=pos)
```

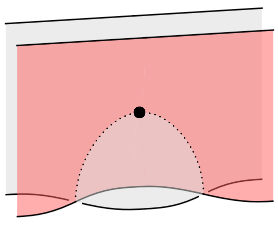
[24]:



And the unbraid 2-cell.

[25]:
```
tgt = l_l @ m_m
b_plus = Cell2(tgt, Rneg << Rpos, cone=0.5)
b_plus_i = Cell2(Rneg << Rpos, tgt, cone=0.5)
b_plus.layout(width=1, height=0.7).render_cvs(pos=pos)
```

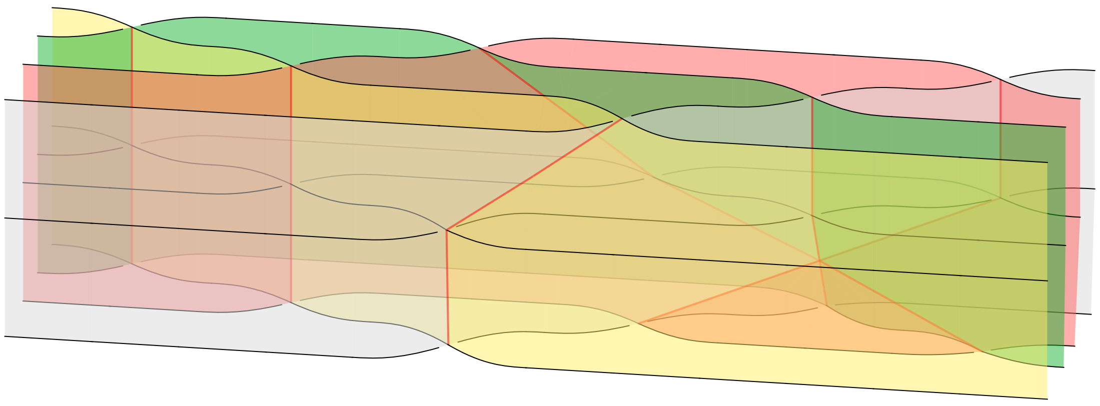
[25]:



A small portion of the Zamolodchikov tetrahedral equation served to stress test the whole system. This example already uses over 1000 variables in the constraint satisfaction.

[26]:
```
alpha = 0.5
green = color.rgb(0.1, 0.7, 0.2, alpha)
blue = color.rgba(0.0, 0.37, 0.90, alpha)
pink = color.rgba(1.0, 0.37, 0.36, alpha)
grey = color.rgba(0.85, 0.85, 0.85, alpha)
cream = color.rgba(1.0, 1.0, 0.92, alpha)
cyan = color.rgba(0.0, 0.81, 0.80, alpha)
yellow = color.rgba(1.0, 0.93, 0.4, alpha)

Cell2.pip_color = None
Cell2.cone = 1.

l = Cell0("l", fill=yellow, st_stroke=st_thin)
m = Cell0("m", fill=green, st_stroke=st_thin)
n = Cell0("n", fill=pink, st_stroke=st_thin)
o = Cell0("o", fill=grey, st_stroke=st_thin)

l_l = Cell1(l, l, pip_color=None, stroke=None)
m_m = Cell1(m, m, pip_color=None, stroke=None)
n_n = Cell1(n, n, pip_color=None, stroke=None)
o_o = Cell1(o, o, pip_color=None, stroke=None)

l_ = l(st_stroke = st_braid+st_thin)
m_ = m(st_stroke = st_braid+st_thin)
n_ = n(st_stroke = st_braid+st_thin)
o_ = o(st_stroke = st_braid+st_thin)

def braid(n, o):
    o_ = o(st_stroke = st_braid+st_thin)
    on_no = Cell1( o_@n , n@o_ , pip_color=None, st_stroke=st_thick+[red.alpha(0.
↪5)])
    return on_no

cell = (l_l @ m_m @ braid(n, o)).extrude()
cell = (l_l @ braid(m, o) @ n_n).extrude() << cell
cell = (
    Cell2((o_o@l_l)<<braid(l,o), braid(l,o)<<(l_l@o_o)) @
    Cell2(braid(m,n)<<(m_m@n_n), (n_n@m_m)<<braid(m,n))) << cell
cell = (o_o @ braid(l,n) @ m_m).extrude() << cell
cell = (o_o @ n_n @ braid(l,m)).extrude() << cell
top = cell

def yb(m,n,o):
    tgt = (o_o @ braid(m,n)) << (braid(m,o) @ n_n) << (m_m @ braid(n,o))
    src = (braid(n,o) @ m_m) << (n_n @ braid(m,o)) << (braid(m,n) @ o_o)
    cell = Cell2(tgt, src)
    return cell

bot = (l_l<<l_l<<l_l).extrude() @ yb(m,n,o)
bot = (
    (o_o @ n_n @ braid(l,m)).extrude() <<
    (o_o @ braid(l,n) @ m_m).extrude() <<
    (braid(l,o) @ n_n @ m_m).extrude() << bot)

top * bot
```

[26]:



7 Cell attributes

Default values for various attributes can be set on the `classes` themselves.

[27]:
```
from huygens.wiggle import black, st_normal, st_Thick

class Atom(object):
    " abstract base class for all cell objects "


class Cell0(Atom):
    "These are the 0-cells, or object's."

    fill = None
    stroke = black
    st_stroke = st_normal
    pip_cvs = None
    on_constrain = None


class Cell1(Atom):
    stroke = black
    st_stroke = st_Thick
    pip_color = black
    pip_radius = 0.06
    pip_cvs = None
    on_constrain = None


class Cell2(Atom):
    pip_color = black
    pip_radius = 0.08
    pip_cvs = None
    cone = 0.6 # closer to 1. is more cone-like
    on_constrain = None

# now remember the real classes
from huygens.wiggle import Cell0, Cell1, Cell2
```

8 Saving output

From any `Cell0`, `Cell1`, or `Cell2` object, we can extract a `Canvas` object which knows how to write a file to disk.

[28]:
```
cell = Cell0("n", fill=pink)
cvs = cell.render_cvs()
cvs.writePDFfile("cell.pdf")
cvs.writeSVGfile("cell.svg")
cvs.writePNGfile("cell.png")
```