Analysis of Mom-4 manifolds

Robert C. Haraway, III[*]

October 26, 2014

This is a literate program to analyze Mom manifolds from [1]. We use a computer and the programs `Regina`, `SnapPy`, `HIKMOT`, and some `Python` modules to accomplish this.

Now, Milley's files once available at the Commentarii website are no longer there. I am going to ask Milley's permission to host them on my website.

These files contain the gluing information for Milley's candidate Mom-4 manifolds. Since Regina is such a nice general program for studying general 3-manifolds, the first order of business is to write a program to turn the gluing data into ideal Regina `NTriangulation`s. Perhaps we should first define ideal triangulation.

Definition 1. An oriented *ideal 3-triangulation* is a space resulting from a orientable face-pairing of oriented solid tetrahedra, such that no face-pairing identifies an oriented edge to itself backwards.

An orientable ideal triangulation *of an orientable 3-manifold* $(M, \partial M)$ is a homeomorphism $\phi : T \setminus T_0 \to M \setminus \partial M$, where T is an oriented ideal 3-triangulation, and T_0 is its vertices.

We note that the link of a vertex of an ideal triangulation corresponds through ϕ to a connected component of ∂M. In particular, closed 3-manifolds have no ideal triangulations. One may, however, express a closed 3-manifold as a Dehn filling of a "parent" manifold, and then ideally triangulate the parent. This is how `SnapPy` represents closed hyperbolic 3-manifolds.

[*]Research partially supported by NSF grant DMS-1006553.

Onward to the translation program: here is the salient quotation from Milley's documentation:

> The input and output files for these programs use my own notation for describing triangulation manifolds, which I'll describe here. As an example, consider the following:
>
> ```
> (33): 11 10 6 7 9 8 2 3 5 4 1 0
> ```
>
> The above string describes a Mom-2 manifold which happens to be the manifold known in the `SnapPea` census as `m203`. Each Mom-n manifold dealt with by these programs is described as a gluing of a set of ideal dipyramids. The *prefix* to the string ("(33):" in this case) describes the set of dipyramids that are used to construct the manifold. In the case of the string above, "(33):" indicates that a pair of 3-sided dipyramids are glued together to obtain the manifold.
>
> The rest of the string describes how the faces of the dipyramids are glued together. Note that an n-sided dipyramid will have n "equatorial" ideal vertices and two "polar" ideal vertices. We always glue faces together in such a way that "polar" vertices are identified with other "polar" vertices. This restriction means there is always only one way to glue to [sic] faces together that correctly preserves orientation. So we need only specify which faces are glued to which faces.
>
> Divide each dipyramid into a "northern" and a "southern" hemisphere, with one "polar" ideal vertex in each hemisphere.

Choose a “northern” face of the first dipyramid and number it 0. Proceed counterclockwise around the “north pole” of this dipyramid, numbering the subsequent faces 1, 2, and so on until you run out of “northern” faces on the first dipyramid. Then label the “northern” faces of the next dipyramid using subsequent integers in the same way, then the next dipyramid, and so on, until all of the “northern” faces have been labelled from 0 to $k-1$ for some k. Then label the “southern” faces from k to $2k-1$, such that face 0 is adjacent to face k, face 1 is adjacent to face $k+1$, and so on until face $k-1$ is adjacent to face $2k-1$.

With all the faces so labelled, the gluing is completely determined by a permutation of length $2k$ consisting of k transpositions. In the example above, the permutation is “11 10 6 7 9 8 2 3 5 4 1 0”. So face 0, for example, which is a “northern” face on the first dipyramid which is directly above face 6, is identified with face 11, a “southern” face on the second dipyramid which is directly below face number 5.

There are 10 possible prefixes. “(4):” and “(33):” give Mom-2’s, “(5):”, “(34):”, and “(333):” give Mom-3’s, and “(6):”, “(35):”, “(44):”, and “(3333):” give Mom-4’s.

First of all, ideal dipyramids, or *dipyrs*, are paramount in this work. So as a warmup, write a program to construct an n-dipyr in Regina. Here is my code:

⟨*construct dipyr*⟩≡
```
def make_dipyr(n):
  """Returns an n-dipyr."""
  newt = regina.NTriangulation()
  for i in range(0,n):
    newt.newTetrahedron()
  for i in range(0,n):
    me = newt.getTetrahedron(i)
    you = newt.getTetrahedron((i+1)%n)
    me.joinTo(2,you,NPerm4(2,3))
  return newt
```

This code chunk illustrates the Python and Regina formalisms that will be used throughout this work. Those unfamiliar with these are missing out, and should visit their respective websites [3] and [4] to learn more.

Here is what this code chunk means, line by line.

`def make_dipyr(n):` means “The following code is a definition for a procedure called `make_dipyr` which takes one argument, which we will call `n`.”

`"""Returns an n-dipyr."""` is a Python-docstring, where “doc” is short for “documentation”. It explains briefly what the procedure does. I say “procedure,” because it is not a function. Every time it is called with argument n, it will make a new n-dipyr.

`newt = NTriangulation()` basically means “Let `newt` start off as an empty triangulation.”

The next two lines basically mean “Let `newt` be the disjoint union of n tetrahedra.”

The next four lines require more explanation.

Let T and B be the top and bottom polar vertices of an n-dipyr, and let $v_i, i \in \mathbb{Z}/n\mathbb{Z}$ be its equatorial vertices in order. By “in order”, I mean that v_i and v_{i+1} are connected by an edge. Then we may triangulate the n-dipyr by a cyclic list of n tetrahedra, the i^{th} element of which is the tetrahedron T_i with vertices $T, B, v_i, v_{(i+1)}$, for $i \in \mathbb{Z}/n\mathbb{Z}$. Equivalently, the i^{th} element of this list is the tetrahedron T_i with vertices $T, B, v_i, v_{(i+1)\%n}$, where $0 \leq i < n$ and $k\%n$ is the least natural number a such that $k \equiv a \mod n$.

Conversely, we may construct an n-dipyr from these n tetrahedra by gluing them up appropriately. The appropriate gluings glue the face TBv_{i+1} in T_i to TBv_{i+1} in T_{i+1}, preserving incidence. The code implements this in Regina. Note that in Regina, the vertices of a tetrahedron are labelled `0,1,2,3`, and the faces are labelled by the vertices they omit.

We know we want to glue T_i to $T_{(i+1)\%n}$ for each $0 \leq i < n$. The last four lines do this. The first line means “For all $0 \leq i < n$, run the following indented code block:”. The next two lines mean “Let `me` be T_i and `you` be $T_{(i+1)\%n}$.”

The last line of the indented code block under the `for` statement means "Glue the face of `me` opposite the vertex `2` to the face of `you` using the gluing map that permutes the vertices by the transposition (2 3)."

The explanation for this is as follows. Vertex `2` of `me` should be $v_{(i+1)\%n}$ in the n-dipyr, and vertex `3` of `you` should also be v_{i+1}. Furthermore, since both `2` and `3` will become equatorial vertices in the n-dipyr, `0`,`1` will become polar vertices. Then the gluing map should send the face `013` (the face opposite `2`) of T_i to the face `012` (the face opposite `3`) of $T_{(i+1)\%n}$. So on vertices, this gluing map acts as the transposition (2 3). This concludes the explanation of the last line of the `for` loop block, and the explanation of the penultimate four lines.

The last line means "The final result of this procedure is the value of `newt`," the value of `newt` being, of course, the newly-constructed n-dipyr, represented as a `Regina NTriangulation`. That concludes my explanation for this code chunk.

This code will not suffice, for Mom-4 manifolds may be glued from multiple dipyrs. So we should implement a procedure to construct a disjoint union of dipyrs according to a Mom prefix. That is, we want a procedure that constructs a `Regina NTriangulation` with an n-dipyr for every n in `prefix`, with multiplicity, where `prefix` is a finite tuple of positive integers. It might be instructive to the reader to write such a procedure in `Regina-Python`. Here is my code:

⟨*make dipyrs by prefix*⟩≡
```
def make_dipyrs(prefix):
  """Return dipyrs specified by prefix."""
  newt = regina.NTriangulation()
  offset = 0
  for i in prefix:
    for j in range(0,i):
      newt.newTetrahedron()
    for j in range(0,i):
      me = newt.getTetrahedron(offset+j)
      jj = (j+1) % i
      you = newt.getTetrahedron(offset+jj)
      me.joinTo(2,you,regina.NPerm(2,3))
    offset += i
  return newt
```

Now we need to write code to glue up the remaining faces of the dipyrs according to the rest of Milley's Mom-strings. The rest of the Mom-string is a permutation in Cayley notation on $2n$ elements, where n is the sum of the prefix. Eventually, we would like to represent this not as a string, but as a $2n$-tuple. Suppose then that `perm` is such a $2n$-tuple. Then the face Milley calls `i` should be glued to the face Milley calls `perm[i]`.

Question: which face of `make_dipyrs(prefix)` is the face Milley calls `i`?

Answer: Face `depth` of tetrahedron `tet_idx`, where `(tet_idx,depth)` is `face(prefix,i)`, and where the latter is given by the following.

⟨*tuples to triangulations*⟩≡
```
def face(prefix,i):
  n = sum(prefix)
  tet_idx = i % n
  if i < n:
    depth = 0
  else:
    depth = 1
  return (tet_idx,depth)
```

Finally, therefore, a Mom manifold specified by one of Milley's Mom-strings is given by the following procedure.

⟨*Mom manifold*⟩≡
```
def make((prefix, perm)):
  newt = make_dipyrs(prefix)
  for i in perm:
    if i < perm[i]:
      ⟨glue up the pair of faces⟩
  label = str((prefix,perm))
  newt.setPacketLabel(label)
  return newt
```

⟨*glue up the pair of faces*⟩≡
```
(tet_i,depth_i) = face(prefix, i)
(tet_j,depth_j) = face(prefix, perm[i])
me = newt.getTetrahedron(tet_i)
you = newt.getTetrahedron(tet_j)
if depth_i == depth_j:
  p = regina.NPerm(2,3)
else:
  p = regina.NPerm(0,1)
me.joinTo(depth_i,you,p)
```

This bears a small amount of explanation. Northern faces have depth 0, and southern faces depth 1. A gluing map (of the sort considered above) between faces of the same depth will send `0 1` to `0 1`. Since the map must be orientation reversing, its action on the vertices can't be the identity. So it must be given by (2 3). Similarly, a gluing of faces with different depth must send `0 1` to `1 0`. It can't also switch `2 3`, for then it would preserve orientation. So it must be given by (0 1).

Having written a program to make a Mom manifold from two tuples representing, respectively, the prefix and permutation of a Mom string, let us now write a function to transform a Mom string into such a pair of tuples. We'll do it in `Regina-Python` since `Python` has good string-manipulation libraries.

⟨*Milley to Regina-Python*⟩≡

```
import shlex
def parse(mill):
  tokenized  = shlex.split(mill)
  perm_str   = tokenized[1:]
  perm       = tuple(map(int,perm_str))
  pref_tok   = tokenized[0]
  a = pref_tok.find("(") + 1
  b = pref_tok.find(")")
  pref_str = pref_tok[a:b]
  prefix = tuple(map(int,pref_str))
  reginafied = (prefix,perm)
  return reginafied
```

`map` is a higher-order function that applies its initial argument to all the elements of the following argument (assumed to be a `list` or `tuple` or some other `Iterable`) and returns the resulting values in a `list`.

In this case, its first argument is the function `int`, which attempts to interpret a string as a number in the usual way—e.g. `int("394")` returns the integer value three hundred ninety four, but `int("Fangorn")` fails.

`shlex.split` regards its argument as a series of *tokens*—space-free strings—separated by spaces. It returns the `list` of these tokens in order.

The first part of this method will need to be changed in the event that the prefix notation starts using spaces to distinguish numbers. Namely, it will need to tokenize `mill_prefix` using `shlex.split` as for `perm`.

`tuple` creates an *immutable* tuple from its argument. Conceptually, this isn't that important; it just bars me from mistakenly changing things down the line. It's programming hygiene.

Now we need to get Milley's data into `Regina-Python`. Fortunately, within each file Milley has conveniently split the data into lines, and `Python` has nice idioms and routines for such files.

After getting the data in, we must check whether each manifold is hyperbolic of finite volume or not (i.e. hyp), and among the hyperbolic ones which are isometric. There is now a `Regina-SnapPy` module `unhyp` for determining whether or not a manifold is hyperbolic. SnapPy can already reliably identify known manifolds and tell whether or not they are isometric.

⟨*vet a file*⟩≡
```
  from unhyp import isHyp
  def hyps_in(file):
    hyps = []
    for line in f:
      if line == "X\r\n":
        break
      print "Working on " + line[0:-1]
      m = make(parse(line))
      m.intelligentSimplify()
      if isHyp(m):
        hyps.append(m)
    return hyps

  def remove_dups(list,eq):
    no_dups = []
    for a in list:
      for b in no_dups:
        if eq(a,b):
          break
      else:
        no_dups.append(a)
    return no_dups

  import snappy
  def vet(file):
    hyps = hyps_in(file)
    for hyp in hyps:
      hyp.finiteToIdeal()
      hyp.intelligentSimplify()
    to_snap = lambda m: \
      snappy.Manifold(m.snapPea())
    snaps = map(to_snap,hyps)
    for snap in snaps:
      snap.canonize()
    idfy = lambda x: x.identify()[0]
    idfys = map(idfy,snaps)
    isom = lambda x,y: \
             x.is_isometric_to(y)
    dupfree = remove_dups(idfys,isom)
    name = lambda x: x.name()
    names = map(name,dupfree)
    names.sort()
    for n in names:
      print n
```

⟨*mom.py*⟩≡
```
  import regina
  ⟨make dipyrs by prefix⟩
  ⟨tuples to triangulations⟩
  ⟨Mom manifold⟩
  ⟨Milley to Regina-Python⟩
  ⟨vet a file⟩
  import sys
  if __name__ == "__main__":
    filename = sys.argv[1]
    f = open(filename)
    vet(f)
```

References

[1] David Gabai, G. Robert Meyerhoff, and Peter Milley, "Mom technology and volumes of hyperbolic 3-manifolds," Comment. Math. Helvetici. **86** (2011) 145–188.

[2] N. Hoffman, K. Ichihara, M. Kashiwagi, H. Masai, S. Oishi, and A. Takayasu, "Verified computations for hyperbolic 3-manifolds," submitted (arXiv:1310.3410), 2013.

[3] Guido van Rossum, et al., `Python`, http://www.python.org/

[4] Benjamin A. Burton, Ryan Budney, William Pettersson, et al., "`Regina`: Software for 3-manifold topology and normal surface theory", http://regina.sourceforge.net/, 1999–2013.