chubby

Robert C. Haraway III

September 7, 2020

Abstract

We enumerate necklace gluings, determine some hyperbolic knot complements into which they embed non-elementarily, and then calculate the exceptional Dehn fillings of those knot complements.

Conventions

We use camel-case naming, following Regina. We remind the reader that indexing begins at 0.

1 Necklace gluings with n beads

Definition 1.1. An n-dipyramid is a 3-ball with a cell structure whose boundary triangulation is the suspension of an n-gon. The choice of suspension points is canonical unless $n = 4$, in which case we take care to remember which vertices are suspension points.

The following is our implementation of this definition in Regina.

⟨*how to make an n-dipyr*⟩≡

```
def makeDipyr(n):
    """Returns an n-dipyr."""
    newt = regina.Triangulation3()
    for i in range(0,n):
        newt.newTetrahedron()
    for i in range(0,n):
        me = newt.simplex(i)
        you = newt.simplex((i+1)%n)
        me.join(2,you,regina.Perm4(2,3))
    return newt
```

As you can verify, we have glued up the n-dipyramid so that the interior faces are all faces 2 and 3 of their respective tetrahedra, and the boundary faces are thus faces 0 and 1. Their common edge in a given tetrahedron is an edge of the base polygon. We have also indexed the tetrahedra with naturals less than n so that the induced cyclic ordering on the common edges is the same as that induced by the base polygon. Finally, as you may verify, all the 0 faces lie around one suspension point, and the 1 faces lie around the other suspension point.

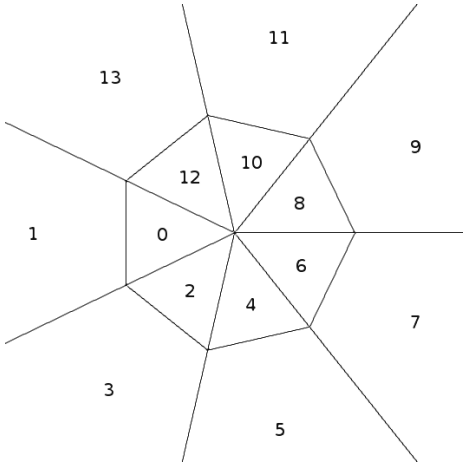

Figure 1: Our labelled 7-dipyramid in stereographic projection from the higher suspension point.

N.B. In the above paragraph, the labels 0,1,2,3 refer to Regina's internal labelling system, not our just imposed face labelling on the boundary faces of the dipyramid.

Definition 1.2. A *necklace gluing of bead number* n is a oriented face-pairing of all the faces of an n-dipyramid such that every face-pair preserves the partition of the dipyramid's vertices into suspension points.

The set of necklace gluings of bead number n on an n-dipyramid is naturally in bijection with the perfect matchings on the set of faces of that n-dipyramid. There are $2 \cdot n$ faces, so given any labelling of the faces by naturals less than $2 \cdot n$, we get an identification sending perfect matchings in $S_{2 \cdot n}$ to necklace gluings.

The labelling we choose involves a choice of higher and lower suspension point, an orientation of the base polygon, and a choice of face thereon. With these choices, mark the sides of the base polygon with $\{2 \cdot i : 0 \leq i < n\}$, such that the chosen face takes marking 0, and the cyclic ordering on faces from the orientation induces the natural cyclic ordering $0 < 2 < \cdots < 2 \cdot n - 2$. Then a face of the n-dipyramid which is adjacent to the lower suspension point takes as a label the marking on the side of the base of which it is a cone, but a face of the n-dipyramid adjacent to the higher suspension point takes instead 1 plus the marking on its base side. (Cf. Figure **??**.)

Therefore, the following code accomplishes a necklace gluing, given a perfect matching in $S_{2 \cdot n}$ in cycle notation, assuming Regina maintains the indexing of tetrahedra.

First, it is easy to match a face label with a tetrahedron index.

⟨*how to make a necklace gluing*⟩≡
```
def whichTet(face):
    return face / 2
```

We represent face-pairings as lists of pairs of natural numbers. To accomplish a gluing, we make an appropriate dipyramid, and for every pair in the list, glue the faces together. The first task is to determine which tetrahedras' faces are getting glued together.

⟨*how to make a necklace gluing*⟩+≡
```
def makeNecklace(match):
    n = len(match)
    ndipyr = makeDipyr(n)
    local = list(match)
    while local != []:
        (i,j) = local.pop()
        me = ndipyr.simplex(whichTet(i))
        you = ndipyr.simplex(whichTet(j))
```

The face of `me` getting glued is either 0 or 1, respectively, according as i is adjacent to the lower or higher suspension point, i.e. according as i is even or odd.

⟨*how to make a necklace gluing*⟩+≡
```
        meFace = i % 2
```

We now split into two cases according as i, j lie on the same or different suspension points.

If they lie along the same suspension point, then the face-pairing restricted to the vertices fixes 0 and 1. (Recall the duality between faces and vertices of a tetrahedron.) But it must reverse orientation, so that the whole manifold is oriented. Thus the induced map on the vertices is the permutation (2 3).

⟨*how to make a necklace gluing*⟩+≡
```
        if (i % 2 == j % 2):
            me.join(meFace,you,regina.Perm4(2,3))
```

On the other hand, if they lie along different suspension points, then the 0,1 vertices must be flipped. Thus the 2 and 3 vertices must be fixed so that the gluing is oriented.

⟨*how to make a necklace gluing*⟩+≡
```
        else:
            me.join(meFace,you,regina.Perm4(0,1))
    ndipyr.setLabel(str(match))
    return ndipyr
```

1.1 Examples of necklace gluings

As examples of necklace gluings, we give the 3-sphere, RP^3, and also our favorite example, the Whitehead link exterior. The reader may pick his or her own favorite two-sided one-relator Heegaard diagram, translate it into a necklace gluing, and test the above construction.

⟨*exampleGluings.py*⟩≡
```
import regina
from necklace import makeNecklace
```

1.1.1 S^3

Uniquely among tame 3-manifolds, S^3 has a genus-0 Heegaard splitting. The easiest diagram for this splitting is with the usual 1-tetrahedron triangulation. This is a necklace gluing, namely the gluing `(0,1)`.

⟨*exampleGluings.py*⟩+≡
```
s3 = makeNecklace([(0,1)])
```

1.1.2 RP^3

Real projective space is the quotient of a ball by the antipodal map. The simplest dipyramid on which this map is a pole-preserving gluing is the 2-dipyramid.

⟨*exampleGluings.py*⟩+≡
```
rp3 = makeNecklace([(0,3),(1,2)])
```

1.1.3 T^3

The 3-torus has a very clear Heegaard splitting coming from its cellulation by a cube in the usual way. Let $k = \mathbf{F}_2$ be the field of two elements. The vertices of the cube are elements of k^3, with edges between vertices differing in one entry, and faces being those vertices with one entry in common. Cellulate the cube by taking the cone off its boundary on an interior point, say the center c. Then the cells are (square-base) pyramids; identifying pyramids along their bases yields three 4-dipyramids. This ends up being a necklace gluing. This gluing therefore has 24 sides. Each side lies in some pyramidal half of a dipyramid. We may identify this side by its lateral edge and the base of its associated pyramid.

⟨*exampleGluings.py*⟩+≡
```
t3facePairings = \
   [(0,14),(1,10),(2,19),(3,23), \
    (4,15),(5,11),(6,18),(7,22), \
    (8,16),(9,20),(12,17),(13,21)]
```

However, this is not a one-relator Heegaard splitting, and so lies outside the scope of this project.

1.1.4 Whitehead link exterior

The following yields the Whitehead link exterior.

⟨*exampleGluings.py*⟩+≡
```
whlx = makeNecklace([(7,3),(6,1),(5,2),(4,0)])

if __name__ == "__main__":
    if not s3.isThreeSphere():
        raise Exception("S3 gluing failed")
    else:
        print "s3 is S3"
    x = regina.Census.lookup(rp3).first().name()
    if x != 'RP3 : #1':
        raise Exception("RP3 gluing failed")
    else:
        print "rp3 is RP3"
    x = regina.Census.lookup(whlx).first().name()
    if x != 'm129 : #2':
        raise Exception("Whitehead exterior failed")
    else:
        print "whlx is Whitehead link exterior m129"
```

⟨*necklace.py*⟩≡
```
import regina
import snappy
⟨how to make an n-dipyr⟩
⟨how to make a necklace gluing⟩
```

2 Enumerating necklace gluings

Our next goal is to get a list as small as possible which for every necklace gluing contains a manifold homeomorphic to that gluing. We could, of course, just enumerate all perfect matchings and put the associated gluings in some container. But many of these, presumably, will be asymmetric, so that by applying elements of the symmetry group of the n-dipyramid, we get different gluings which are homeomorphic. We wish to eliminate as much of such redundancy as we reasonably can.

A reasonable approach is to have our container contain one representative from each dipyr-symmetry group orbit of perfect matchings, and nothing else. Milley took this approach, and it is the approach we take as well. However, we introduce two ideas which makes the enumeration faster and leaner.

The first idea is that we don't care so much about the particular Heegaard splitting we get from necklace gluings, and are concerned primarily with homeomorphism. Isomorphism of triangulations is a less restrictive equivalence relation on triangulations which still only equivalates homeomorphic manifolds. So we don't mess about with group orbits, and just look at isomorphism signatures, which are conveniently already implemented in both Regina and SnapPea. This was not available to Gabai, Meyerhoff, and Milley during their work.

The second idea is as follows. We begin by constructing a trie with a face-pair at each internal node, such that there is an identification of perfect matchings in $S_{2\cdot n}$ with backtrack-free paths from the root of the trie to leaves. This takes much less space to store than the whole list of perfect matchings. We define what it means to remove a matching from the trie. This takes much less time than removing a matching from a list. Finally, we do a fairly natural thing after that: while the trie still contains a matching σ, calculate its isomorphism signature, put that signature in a Python `set`, and remove σ from the trie. This yields a set of distinct isomorphism signatures that exhaust the isomorphism classes of elements in the trie.

Remark 2.1. There are several other optimizations and generalizations possible. In particular, one can perform a depth first search on the above trie without constructing it first. One could also perform tests at junctures in the computation to determine if all the descendant leaves of a node are invalid for some reason and can be discarded—this is called *pruning*. In pruning, one would probably want to reintroduce the orbit considerations used by Gabai, Meyerhoff, and Milley.

However, the above approach, implemented below, is sufficient for our needs, and is relatively simple to understand. Nevertheless, we have implemented it in such a way that later one should be able to reimplement this enumeration without affecting the other portions of the program—in particular, the final computation's implementation would remain unchanged.

2.1 The trie of gluings

We implement the trie of gluings as a standard binary trie on pairs of natural numbers. Its type's definition in Coq would be

⟨*Coq definition of the type of our trie*⟩≡
```
Inductive PairTrie : Type :=
| PTLeaf : PairTrie
| PTBranch : (nat*nat) -> PairTrie -> PairTrie -> PairTrie.
```

That is to say, a pair-trie is either a leaf, or it's a branch, with an associated pair (m, n) of numbers and two associated pair-tries, a left child and right child. We again go with a poor-man's implementation of this datatype in Python. A pair-trie will be a tuple, either a singleton or a quadruple. If it is a singleton, its single entry will be the constant `PTLeaf`. Otherwise, its initial entry will be the constant `PTBranch`; its next entry will be a pair `(m,n)` of integers; and its final entries will be pair-tries.

The way we construct our trie follows from a more general construction of perfect matchings on arbitrary lists of even length. Suppose $pfm(l)$ were the set of perfect matchings on l. If l is empty, then this set is empty; we thus represent it by a leaf. Otherwise, l has at least two elements by evenness. We partition $pfm(l)$ according as an element does or does not flip the first two elements of l. Thus, $pfm(l) = (f\ f') \cdot pfm(l'') \cup X$, where f, f' are the first elements of l; l'' is l without these elements; and X is the subset of $pfm(l)$ sending f to something apart from f'.

At this point we end up with two options. Either we could use a non-binary trie, or we can introduce a sort of auxiliary stack parameter in pfm to "remember" which face-pairs we've foregone so far. We choose the latter option.

So we end up with a slightly more complicated decomposition. We have some set m, and $pfm(l, m)$ is going to be the set of perfect matchings on $l \cup m$ *which sends the first element of* l *to some other element of* l. (Thus, our original definition of pfm would now be $pfm(l, \emptyset)$.) If l is empty, this is $\emptyset$. We represent this situation by a leaf. Otherwise, by l's evenness, it has two initial elements f, f'. Then we may decompose $pfm(l, m)$ into those matchings which flip f, f' and those which don't. We may write a perfect matching on $l \cup m$ which flips f, f' as the composition of $(f\ f')$ and a perfect matching of $l'' \cup m$, where l'' is l without f, f'. On the other hand, a perfect matching on $l \cup m$ which does not send f to f' but which does send f into l is a perfect matching on $l' \cup (\{f'\} \cup m)$ which sends f into some other element of l', where l' is l without f'. Thus we have a particular pair (f, f'), and two similar, simpler pfms to consider. We represent this situation by a branch. (For convenience, we define an auxiliary "smush" function establishing union in the first part of the partition.)

⟨*how to make the gluing trie*⟩≡
```
  def smush (l,m):
      if m == []:
          return l
      else:
          return smush (([m[0]] + l), m[1:])

  PTLeaf = -1
  PTBranch = -2

  def pfm(l,m):
      if l == []:
          return (PTLeaf,)
      # else
      f = l[0]
      lp = l[1:]
      if lp == []:
          return (PTLeaf,)
      fp = l[1]
      lpp = l[2:]
      return (PTBranch, \
              (f,fp), \
              pfm(smush(lpp,m), []), \
              pfm([f] + lpp, [fp] + m))

  def completePairTrie(n):
      nums = range(2*n)
      nums.reverse()
      return pfm(nums, [])
```

It's pretty simple to extract at least one matching from our trie if there is one. Just go left.

⟨*how to get a matching if there is one*⟩≡
```
  def leftMatching(pt):
      if pt == (PTLeaf,):
          return []
      else:
          (str, p, lt, rt) = pt
          assert str == PTBranch
          return [p] + leftMatching(lt)
```

It's also fairly straightforward to remove a matching from our trie. This, however, requires now that we spell out exactly what is the association between a path from root to leaf and perfect matching. By cycle notation it will suffice to describe how to extract from such a path a sequence of n face-pairs (in order to get a permutation in cycle notation). (The involutive and perfect properties must be proved from our original trie's properties, which proof we leave to the reader.)

A path from root to leaf might be empty; in that case, the sequence is empty. Otherwise, the path takes some sequence of lefts and rights. We say that the sequence of a path is that list whose first element is the pair of the node at which the path first turns left, and whose tail is the associated sequence of the path from the root of the child trie to the given leaf.

The application of this for removing a matching is as follows. One has given a matching in the proper order, and a pair-trie again in proper order. (So if we were being thorough, we would need to define proper order and show the original trie is properly ordered, and show that removal preserves propriety.) If the trie is empty, you return the trie. Otherwise, compare the root with the head of the matching. If they differ, remove the whole matching from the right child. If, instead, they agree, then remove the rest of the matching from the left child. If the result is just a leaf, then we can get rid of this root entirely, so the result should be the right child. Otherwise, it's just the trie with the rest of the matching removed from the left child.

⟨*how to remove a matching*⟩≡
```
def removeMatching(l,pt):
    if l == []:
        return pt
    if pt[0] == PTLeaf:
        return pt
    p,ps,(str,pp,lt,rt) = l[0],l[1:],pt
    assert str == PTBranch
    if p != pp:
        return (PTBranch, pp, lt, \
                removeMatching(l,rt))
    ltp = removeMatching(ps,lt)
    if ltp[0] == PTLeaf:
        return rt
    else:
        return (PTBranch, pp, ltp, rt)
```

This takes time proportional to at most the number of face-pairs. If we had kept the matchings in a list, removing an arbitrary matching would take time proportional to the number of *matchings*!

In conclusion for this section, it's now completely straightforward to implement our original plan as described above: construct the trie, and while the trie is not a leaf, pop the leftmost matching and record its isomorphism signature.

⟨*how to enumerate isosigs of necklaces*⟩≡
```
def necklaceIsoSigs(n):
    sigs = set()
    tr = completePairTrie(n)
    while tr != (PTLeaf,):
        sigma = leftMatching(tr)
        tr = removeMatching(sigma,tr)
        x = makeNecklace(sigma)
        sigs.add(x.isoSig())
    return sigs
```

We can now generate all the manifolds of interest to us: necklace gluings of bead number at most 7.

2.1.1 Tests

To test the enumeration we run it on sets of even cardinality at most 14. This runs fairly quickly. Clearly any further optimization here is unnecessary for our purposes.

⟨*enumeration testing*⟩≡
```
if __name__ == "__main__":
    for i in [4,5]:
        all = []
        print "Computing the complete pair trie for " + str(2*i) + " elements..."
        pt = completePairTrie(i)
        print "Here are all the matchings:"
        j = 0
        while pt != (PTLeaf,):
            j += 1
            x = leftMatching(pt)
            print str(x)
            all.append(x)
            pt = removeMatching(x,pt)
        print "That's " + str(j) + " matchings."
        print "Now for the isomorphism signatures."
        sigs = necklaceIsoSigs(i)
        for sig in sigs:
            print str(sig)
        print "That's " + str(len(sigs)) + " isomorphism signatures."
        print "\n"
```

⟨*enumerateGluings.py*⟩≡
```
from necklace import makeNecklace
⟨how to make the gluing trie⟩
⟨how to get a matching if there is one⟩
⟨how to remove a matching⟩
⟨how to enumerate isosigs of necklaces⟩
⟨enumeration testing⟩
```

3 Hyperbolicity testing for link complements

Naively, our next task is to determine which among the necklace gluings of bead number at most 7 embed nonelementarily into hyperbolic 3-manifolds.[1] As a first step, we describe how to determine whether or not a given nontrivial link complement is hyperbolic. As a very first step, we can implement some basic sanity checks. Then we implement normal surface routines that find interesting surfaces of non-negative Euler characteristic, which for link complements are the only obstructions to hyperbolicity, by Thurston's hyperbolization theorem for Haken manifolds.

3.1 Sanity checks

3.1.1 Link complement testing

All necklace gluings are orientable. Likewise, they are all connected. (If we were testing arbitrary MOM gluings, then we would add a connectedness test.) One other requirement, though, is that the links of all the vertices must be tori. This is known in the literature as a "(generalized) link complement," i.e. the complement of a nontrivial link in some closed orientable 3-manifold. (Some authors require that the given closed 3-manifold be irreducible. We do not make this requirement.)

⟨*link complement*⟩≡

```
def isLinkComplement(mfld, allowInteriorVertices):
    if not mfld.isOrientable():
        return False
    if not mfld.isConnected():
        return False
    M = regina.Triangulation3(mfld)
    M.finiteToIdeal()
    if not M.isValid():
        return False
    vs = mfld.vertices()
    for v in vs:
        if not (v.link() == v.TORUS or \
          (allowInteriorVertices and \
           v.link() == v.SPHERE)):
            return False
    else:
        return True
```

3.1.2 Fundamental group obstructions

Next we describe some obstructions to the hyperbolicity of a 3-manifold M that are often easy to detect using `Regina`. These are presentations of the fundamental group of the forms $\langle a, b \mid \ldots, [a^p, b^q], \ldots\rangle$ or $\langle a, b \mid \ldots, a^p b^q, \ldots\rangle$. We call these *common axis relations*, the first being a common axis *commutator*, and the latter a common axis *equation*.

[1] We end up doing something slightly weaker but still sufficient.

Lemma 3.1. *Presentations with common axis relators do not present fundamental groups of hyperbolic link complements.*

Proof. Suppose such a presentation presents G. Suppose $G < PSL_2\mathbf{C}$.

If P has a common axis commutator or equation, then a^p and b^q commute. Since $G < PSL_2\mathbf{C}$, a and b must commute as well, since commuting elements have the same axis, and a^p, b^q have the same axes as a, b, respectively. Thus $p = \pm q = 1$. So either G is trivial or is $\mathbf{Z} \oplus \mathbf{Z}$, and is not the fundamental group of a hyperbolic link complement. □

⟨*fundamental group obstructions*⟩≡
```
def isCommonAxisCommutator(expr):
    l = expr.terms()
    if len(l) != 4 or \
       l[0] != l[2].inverse() or \
       l[1] != l[3].inverse():
        return False
    else:
        return True

def isCommonAxisEquation(expr):
    l = expr.terms()
    return len(l) == 2

def hasCommonAxisObstruction(mfld):
    G = mfld.fundamentalGroup()
    G.intelligentSimplify()
    if G.countGenerators() == 2:
        for i in range(G.countRelations()):
            if isCommonAxisCommutator(G.relation(i)):
                return (True, "$\\pi_1 = \\langle a,b\\ |\\ \ldots, [a^p,b^q], \ldots \\rangle$")
            if isCommonAxisEquation(G.relation(i)):
                return (True, "$\\pi_1 = \\langle a,b\\ |\\ \ldots, a^p b^q, \ldots \\rangle$")
    return (False,'')
```

3.1.3 Census obstructions

The second class of obstructions come from census checks.

Remark 3.2. We would like to emphasize the following in a remark: the following code depends critically on using the censuses in Regina version *5.1*. We have incorporated this at the beginning of the code; if run in a different version of Regina, it will return no census hit name.

⟨*census obstructions*⟩≡
```
def censusNE(mfld):
    if regina.versionString() != "5.1":
        print "Censuses not vetted for Regina versions not equal to 5.1"
        return None
```

Name	Homeomorphism type
S3	S^3
L(p,q)	$L(p,q)$
S2 x S1	$S^2 \times S^1$
RP3	$\mathbf{R}P^3$
SFS [surf: (a0,b0),...]	$M(S; a_0/b_0, \ldots)$
T x S1	T^3, the three-torus
KB\n2 x~ S1	$K^2\tilde{\times}S^1$, the twisted I-bundle over K^2
T x I / [a b | c d]	T^2 mapping torus with monodromy $\begin{pmatrix} a & b \\ c & d \end{pmatrix}$
sfs0 U/m1 ... U/mx sfsx	Seifert-fibered spaces $sfs_0, \ldots, sfs_x$ identified along their boundaries via homeomorphisms given by m_i (usually $m_1 =$ m and $m_2 =$ n)
Hyp_V	Hyperbolic manifold of volume V
x\#\cdots\#	SnapPea census manifold

Table 1: Names for homeomorphism types in Regina 5.1.

Now, Regina contains several censuses of triangulations. Checking whether or not a triangulation lies in a census is a very fast check—it's essentially linear in the triangulation. Regina assigns "names" to census "hits," i.e. occurrences of a triangulation in a census.

⟨*census obstructions*⟩+≡
```
ch = regina.Census.lookup(mfld)
if ch.first() == None:
    return None
name = ch.first().name()
```

There is apparently no online documentation of the types of names in Regina's censuses. As of version 5.1, an inspection of their names in the Regina GUI reveals that all the names of census manifolds are of the following types.

From the closed orientable census, the names have the form `homeo : #x` , where `homeo` is a string denoting the manifold's homeomorphism type, and `x` is some decimal place-value number used for distinguishing non-isomorphic triangulations of the same manifold. The homeomorphism type strings are as in Table **??**.

All the other orientable censuses are of hyperbolic 3-manifolds or knot or link complements in S^3. The Hodgson-Weeks census has entries $\rho : N$ where ρ is a decimal approximation of the hyperbolic volume and N is an expression of the manifold as a Dehn filling of some SnapPea census manifold. The cusped census has the usual SnapPea census names. The knot and link census has the usual link names.

The following fact follows from the above.

Remark 3.3. In *Regina 5.1* the only census manifolds that embed nonelementarily in hyperbolic link complements are themselves hyperbolic.

Implementing a census check for hyperbolic manifolds, and indeed for nonelementary embedding, is almost completely straightforward. We just have to distinguish the two types of name beginning with L, viz. lens spaces and knot and link complements in S^3.

⟨*census obstructions*⟩+≡
```
if name.find("Hyp") != -1:
    return (True, name)
try:
    vol = float(name[:name.find(':')])
    return (True, name)
except:
    pass
if name[0] in set(['m','s','v']):
    return (True, name)
if name[0] == 'L':
    if name[1] == '(':
        return (False, name)
    else:
        return None
return (False, name)
```

⟨*sanity.py*⟩≡
```
import regina
import snappy
⟨link complement⟩
⟨fundamental group obstructions⟩
⟨census obstructions⟩
```

3.2 Finding faults

Now comes the first difficult computation. We introduce some nonstandard terminology here.

Definition 3.4. A *fault* in a nontrivial link complement M is a properly embedded surface with nonnegative Euler characteristic that is either

- nonorientable;
- a sphere not bounding a ball;
- a disc not separating a ball from M;
- an incompressible torus not parallel to ∂M; or
- an incompressible, ∂-incompressible annulus not parallel to ∂M.

The motivation behind the word "fault" is twofold. First, my background is in hyperbolic manifolds; I regard hyperbolic manifolds as better than others, and so obstructions to hyperbolicity are defects. But a better reason is that if one tries to solve Thurston's gluing equations using Jeff Weeks's methods in SnapPea for a triangulation $\mathcal{T}$ of a nonhyperbolic link complement, then the structure approaches a degenerate structure. I imagine the degenerating structures as stretching some tetrahedra into ever thinner, longer tetrahedra. If they were made of some physical, ductile material, eventually they would snap. Such snap points are called "faults" in materials science. (And also in geology; this is where Thurston's terminology of earthquakes and faults in *surfaces* comes from.) The conglomeration of such faults in a triangulation often seem to "follow along" essential surfaces of high Euler characteristic—hence the name. (This is related to the duality between angle structures and normal surfaces given by the combinatorial area pairing.)

To find such surfaces, we can appeal to normal surface theories, of which there are several implemented in Regina. The spirit of normal surface theory is composed of the following two ideas:

- Normal surfaces are identified with solutions to an integer linear programming problem defined using a triangulation.
- If there is an essential surface, then there is an essential normal such surface among a finite computable set of basic normal surfaces, for some definition of *basic*.

Normal surface algorithms run as follows: enumerate the finite computable set, then check for essential surfaces of a given type in the finite set. If there is no essential surface in the set of given type, then there is no such essential surface in the manifold at all.

The essential surface checks themselves are implemented using normal surface algorithms. And all these algorithms take a relatively long time to run. So it behooves us to implement as many heuristics as possible to skip the enumerations for essential surface checks. This desire informs the following algorithms throughout. As a first step, we can perform the following basic checks to immediately show a surface is not essential.

⟨*trivially inessential*⟩≡
```
def triviallyInessential(surf):
    if surf.isVertexLinking() or \
       surf.isSplitting() or \
       surf.isThinEdgeLink()[0] != None or\
       surf.isThinEdgeLink()[1] != None:
        return True
    else:
        return False
```

3.2.1 Spheres, projective planes, and discs

We first need to determine whether or not the given 3-manifold is reducible. Regina already has a routine for this, but that routine is only guaranteed to work for closed 3-manifolds. So we roll our own, slow version here. It is an interesting question whether or not one could get a (comparatively) much quicker scheme using only quad-vertex surfaces on ideal triangulations. We do not presume to determine this one way or the other.

We have a choice of either using standard coordinates and fundamental surfaces, in which case we can use ideal triangulations (see [**?**]); or we can use quad coordinates and vertex surfaces, but we can only use "material" triangulations. The latter have more tetrahedra. We use the more familiar standard and fundamental surfaces when we can.

The plan is simply to go through the standard fundamental surfaces and check if they are essential spheres or P^2s.

Since the construction of normal surface lists is costly, we do not write functions determining whether or not a 3-manifold has an essential surface of a given type, but rather whether a given normal surface list has a such a surface.

⟨*essential P2*⟩≡
```
def essentialP2In(nsl):
    l = nsl
    n = l.size()
    for i in range(0,n):
        s = l.surface(i)
        x = s.eulerChar()
        if x != 2:
            if x != 1:
                continue
            if s.isOrientable():
                continue
```

At this point we know $s = P^2$. We return its index in the normal surface iterator `l`, and indicate that it is P^2. If there is a projective plane, it is essential ([**?**], Lem. 5.1).

⟨*essential P2*⟩+≡
```
            return (i,"$P^2$")
```

After the `for` loop, if we found no projective planes, then there are none whatever, by normal surface theory.

⟨*essential P2*⟩+≡
```
    else:
        return None
```

Next is the routine for finding essential spheres.

⟨*essential S2*⟩≡
```
def essentialS2In(nsl):
    l = nsl
    n = l.size()
    for i in range(0,n):
        s = l.surface(i)
        x = s.eulerChar()
        if x != 2:
            continue
        if triviallyInessential(s):
            continue
```

If the Euler characteristic is 2, then s is a sphere. Cut M along s into m. (It seems unfortunate here that we cut instead of crush. We should probably be able to crush instead. We leave this optimization for future work; cutting suffices for us.) Then m has two sphere boundary components. We cap them off by idealizing m.

We note here that Python includes indentation as a critical part of its syntax. It's somewhat hard to tell in this document that the following code is indented outside the previous large `if` statement but inside the overall `for` loop. The statements inside the `if` are at indentation level at least 3; the following is at indentation level 2.

⟨*essential S2*⟩+≡
```
        m = s.cutAlong()
        m.finiteToIdeal()
        m.intelligentSimplify()
```

If m is connected, then s was non-separating, and hence essential.

⟨*essential S2*⟩+≡
```
        if m.isConnected():
            return (i, "non-separating $S^2$")
```

Now, suppose instead that s separates M. Since we assumed M was connected to begin with, it separates M into two components. The sphere s is essential precisely when neither component of M cut along s is a ball. We've idealized the cut, so this is equivalent to neither component of m being S^3.

⟨*essential S2*⟩+≡
```
        m.splitIntoComponents()
        m0 = m.firstChild()
        m1 = m0.nextSibling()
        if not (m0.isThreeSphere() or \
                m1.isThreeSphere()):
            return (i,"Connect-sum $S^2$")
```

The following is at indentation level 1, outside the `for` loop. This code is only run if none of the above `return` statements are encountered. In that case, there are no essential spheres or projective planes whatever by normal surface theory, and the manifold is irreducible.

⟨*essential S2*⟩+≡
```
    return None
```

For our sakes, developing a method to find an essential disk *in* M is trivial, for M is assumed to be an irreducible nontrivial link complement. In that case, M admits a compressing disc if and only if it is a solid torus, and this already has an optimized implementation in Regina, viz. the method `isSolidTorus`. The reader may wonder why we use this instead of the more obvious `hasCompressingDisc`. The reason is that, assuming irreducibility, the `isSolidTorus` test can yield `False` more quickly than `hasCompressingDisc` can, since, among other things, `isSolidTorus` does homology checks.

Remark 3.5. We particularly urge the reader to note, however, that we must run `hasCompressingDisc` instead of `isSolidTorus` after cutting along a torus. A torus in an irreducible 3-manifold can cut it into pieces one of which is reducible. Such a torus is called a *convolutube* in the literature; it is the boundary of a knot complement embedded in a 3-ball in the manifold, as a ball with removed knotted arc. We urge the reader to alert or even castigate the author if he has mixed up these two routines in what follows; such confusion has happened multiple times in the past.

You have been forewarned.

3.2.2 Finding essential tori and Klein bottles

First, no hyperbolic nontrivial link complement has an embedded Klein bottle. Any Klein bottle whatever is a witness against hyperbolicity (for nontrivial link complements). We note that if we do return a K^2, then it may not necessarily be an essential K^2.

⟨*essential K2*⟩≡
```
def essentialK2In(nsl):
    l = nsl
    n = l.size()
    for i in range(n):
        s = l.surface(i)
        x = s.eulerChar()
        if x != 0:
            continue
        if not s.isCompact():
            continue
        if not s.isOrientable():
            return (i,"$K^2$")
```

If the `for` loop has concluded, then there is no Klein bottle.

⟨*essential K2*⟩+≡
```
    return None
```

An *essential torus* is an embedded torus that is neither compressible nor boundary parallel. Compressibility is already implemented in Regina. In general, boundary parallelism can be determined using the methods of Jaco and Tollefson; however, these have not yet been implemented in Regina. Nevertheless, for tori, it only requires a test of homeomorphism to $T^2 \times I$.

Neil Hoffman and I have found the following faster method to test for homeomorphism to $T^2 \times I$, and it also only uses methods already implemented in Regina. The method works due to results on knots in solid tori due to Gabai (and also Berge), which we indicate below.

The method in practice is quite simple. We first do some sanity checks. In particular, if we are so fortunate to have a presentation of the fundamental group that Regina recognises [sic] as $\mathbf{Z}^2$, then the manifold is certainly $T^2 \times I$ (by Rolfsen 10.6 and the fact that we have two boundary components). After the sanity checks, we at least know the manifold has two boundary components, both tori. Change the triangulation until the induced triangulation T on some boundary torus has one vertex. (Here we assume Regina will take care of this for us in its simplification routines. It makes no guarantees like this, so we check that it works. If it fails, we fail. This shouldn't ever happen for our simple examples. A more general approach is ongoing work with Neil Hoffman implementing *inflations* due to Jaco and Rubinstein.)

Closing the book along any edge e in T accomplishes a Dehn filling along the flip of e in T. If M is $T^2 \times I$, then every such filling must be a solid torus. (We can think of a Dehn filling of $T^2 \times I$ as an attachment of $T^2 \times I$ to $D^2 \times S^1$ instead of the other way round. Then the $T^2 \times I$ is clearly just a boundary collar.) Conversely, if any such filling is not a solid torus, then M is demonstrably not $T^2 \times I$. As mentioned above, we can check this with Regina.

If we fold along one edge and get a solid torus, then we know M is the exterior of a knot in a solid torus. Folding along two other edges and getting a solid torus puts M in a very special class of manifolds by Gabai's work. Finally, the actual slopes we fill along when closing the book have intersection number 3 with each other. No such manifold, apart from $T^2 \times I$, admits three such fillings. So if we get three solid tori from the above Dehn fillings, then in fact M must be $T^2 \times I$. (Regina's simplification procedures are so effective that we suspect this point in the procedure will not be reached for the manifolds we encounter. Nevertheless we may as well add it for posterity.)

⟨*is T2xI*⟩≡
```
def isT2xI(mfld):
    idl = regina.Triangulation3(mfld)
    idl.finiteToIdeal()
    idl.intelligentSimplify()
    if not (isLinkComplement(idl,True) and \
        len(mfld.boundaryComponents()) == 2):
        return False
    if not mfld.homologyH1().detail() == '2 Z\n':
        return False
    if mfld.fundamentalGroup().recogniseGroup() == '2 Z':
        return True
    M = regina.Triangulation3(mfld)
    M.idealToFinite()
    M.intelligentSimplify()
    t = M.boundaryComponents()[0]
    if t.countFaces(0) != 1:
        raise Exception("weird boundary component")
    edges = t.faces(1)
    for e in edges:
        i = e.index()
        Mp = regina.Triangulation3(M)
        ep = Mp.face(1,i)
        Mp.closeBook(ep,False,True)
        if not Mp.isSolidTorus():
            return False
    else:
        return True
```

Now we look for essential tori.

⟨*essential T2*⟩≡
```
def essentialT2In(nsl):
    l = nsl
    n = l.size()
    for i in range(n):
        s = l.surface(i)
        x = s.eulerChar()
        if x != 0:
            continue
        if not s.isCompact():
            continue
        if not s.isOrientable():
            continue
        if triviallyInessential(s):
            continue
```

Having located a fundamental normal torus, we should test whether or not it is essential. This boils down to whether or not it cobounds a ∂-reducible 3-manifold (implying it is compressible), or a $T^2 \times S^1$ (implying it is boundary-parallel).

Of first concern to us is whether or not it separates. If it doesn't, then it is a non-separating torus and hence essential.

⟨*essential T2*⟩+≡
```
m = s.cutAlong()
m.intelligentSimplify()
if m.isConnected():
    return(i,"non-separating $T^2$")
```

Having found a separating torus, we can now test whether or not it is essential, by testing whether the components of m are either ∂-compressible or $T^2 \times I$.

⟨*essential T2*⟩+≡
```
m.idealToFinite()
m.intelligentSimplify()
m.splitIntoComponents()
m0 = m.firstChild()
m1 = m0.nextSibling()
if m0.hasCompressingDisc() \
   or m1.hasCompressingDisc() \
   or isT2xI(m0) or isT2xI(m1):
    continue
```

If they are not, then the torus is essential.

⟨*essential T2*⟩+≡
```
return (i,"essential $T^2$")
```

Otherwise, if there is no essential torus, then return nothing. (This else is outside the `for` loop.)

⟨*essential T2*⟩+≡
```
    else:
        return None
```

3.2.3 Essential annuli and Möbius strips

Now, as for the atoroidal remainder, there will be hyperbolic and Seifert-fibered manifolds. We can distinguish the Seifert-fibered manifolds by finding essential annuli and Möbius strips in them, assuming the manifolds are nontrivial link complements.

As we are finally treating bounded surfaces, we can no longer use ideal triangulations; the theory of spun-normal surfaces is not yet well-enough developed. However, we can just look among the Q-vertex surfaces in a material triangulation.

Proposition 3.6. *Let T be a material triangulation of an irreducible ∂-irreducible atoroidal link complement M. If M admits an essential annulus, then T admits a Q-vertex essential annulus.*

Proof. Since M admits an essential annulus, by Corollary 6.8 of Jaco-Tollefson T admits a vertex essential annulus A or torus—but it must be an annulus, as M is atoroidal. By the proof of Theorem 2 in Tollefson, A is isotopic to a Q-vertex surface. □

We don't use this just yet, though, to keep parallelism among all the fault-finding routines. We will plug Q-vertex lists into the following functions instead of standard fundamental lists.

⟨*essential M2*⟩≡
```
  def essentialM2In(nsl):
      l = nsl
      n = l.size()
      for i in range(n):
          s = l.surface(i)
          if not s.hasRealBoundary():
              continue
          x = s.eulerChar()
          if x != 0:
              continue
```

At this point we know `s` is either an annulus or a Möbius band. If it's not orientable, it's the latter, and is a witness against hyperbolicity.

⟨*essential M2*⟩+≡
```
          if not s.isOrientable():
              return (i,"$M^2$")
```

After the `for` loop, there were no Möbius bands. So there isn't one in the list.

⟨*essential M2*⟩+≡
```
      else:
          return None
```

Now we look for essential annuli.

⟨*essential A2*⟩≡
```
  def essentialA2In(nsl):
      l = nsl
      n = l.size()
      for i in range(n):
          s = l.surface(i)
          if not s.hasRealBoundary():
              continue
          x = s.eulerChar()
          if x != 0:
              continue
          if not s.isOrientable():
              continue
          if triviallyInessential(s):
              continue
```

At this point we know `s` is an annulus. To determine whether or not it is essential, cut along it. If the complement is connected, then the annulus is nonseparating, and is thus essential.

⟨*essential A2*⟩+≡
```
          m = s.cutAlong()
          if m.isConnected():
              return (i,"non-separating $A^2$")
```

Otherwise, `m` is disconnected. Split it into its components. The annulus `s` is essential when neither of these components is a ball or is a solid torus. (This assertion should be regarded with the same level of suspicion as the assertions about essential tori touted in Remark **??**.)

⟨*essential A2*⟩+≡
```
m.splitIntoComponents()
m0 = m.firstChild()
m1 = m0.nextSibling()
if m0.isBall() \
   or m0.isSolidTorus() \
   or m1.isBall() \
   or m1.isSolidTorus():
   continue
else:
    return (i,"essential $A^2$")
```

Finally, if we have gone through all quad-vertex surfaces and found no essential annulus (or Möbius band), then there is no such surface by Q-normal surface theory.

⟨*essential A2*⟩+≡
```
else:
    return None
```

⟨*faults*⟩≡
⟨*is T2xI*⟩
⟨*essential P2*⟩
⟨*essential S2*⟩
⟨*essential K2*⟩
⟨*essential T2*⟩
⟨*essential M2*⟩
⟨*essential A2*⟩

3.2.4 Packaging fault finding

With all the fault-finding routines done, we can now package them all together into a fault-finding routine. Technically, the final routine is not a hyperbolicity routine, but a faultlessness routine. For nontrivial link complements this is no distinction, by Thurston's hyperbolization theorem. But we shall have occasion to run this on closed manifolds, so we call it `isFaultless` and not `isHyperbolic`.

⟨*how to find a fault the hard way*⟩≡

```
  def findFault(mfld):
      assert isLinkComplement(mfld,True)
      M = regina.Triangulation3(mfld)
      M.finiteToIdeal()
      M.intelligentSimplify()
      l = regina.NormalSurfaces.enumerate(M,regina.NS_STANDARD,regina.NS_FUNDAMENTAL)
      p2 = essentialP2In(l)
      if p2 != None:
          return p2
      s2 = essentialS2In(l)
      if s2 != None:
          return s2
      if mfld.isSolidTorus():
          return ([], "D2")
      k2 = essentialK2In(l)
      if k2 != None:
          return k2
      t2 = essentialT2In(l)
      if t2 != None:
          return t2
      if mfld.isClosed():
          return None
      M = regina.Triangulation3(mfld)
      M.idealToFinite()
      M.intelligentSimplify()
      l = regina.NormalSurfaces.enumerate(M,regina.NS_QUAD,regina.NS_VERTEX)
      m2 = essentialM2In(l)
      if m2 != None:
          return m2
      a2 = essentialA2In(l)
      if a2 != None:
          return a2
      return None

  def isFaultless(mfld):
      if mfld.hasStrictAngleStructure():
          return (True, "strict angle structure")
      s = findFault(mfld)
      if s != None:
          return (False, s)
      else:
          return (True, "no faults")
```

3.2.5 Testing

These routines should find no faults whatever on our examples, except for RP^3, nor indeed should they find faults in any of the hyperbolic census manifolds. They should find faults in the census manifolds admitting such surfaces. We include a few such census manifolds in the following tests.

⟨*fault testing*⟩≡
```
if __name__ == "__main__":
    from exampleGluings import *
    if not isFaultless(s3)[0]:
        raise Exception("essential surface in S3")
    else:
        print "s3 is faultless"
    if isFaultless(rp3)[0]:
        raise Exception("missed RP2 in RP3")
    else:
        print "rp3 is not faultless"
    if not isFaultless(whlx)[0]:
        raise Exception("fault in Whitehead link complement")
    else:
        print "Whitehead link complement is faultless"
```

The first census example is a Seifert-fibered space over an orbifold with four cone points. This admits an essential torus.

⟨*fault testing*⟩+≡
```
    ex0 = regina.Triangulation3('gLALQbccefffemkbemi')
    if isFaultless(ex0)[0]:
        raise Exception("SFS with 4 cone points missing torus")
    else:
        print "SFS with 4 cone points has fault"
```

The next example is two Seifert-fibered spaces on discs with two cone points glued along their torus boundary components, in a way disagreeing with their fiber structure.

⟨*fault testing*⟩+≡
```
    ex1 = regina.Triangulation3('hLAvQkbccefggglpksulwb')
    if isFaultless(ex1)[0]:
        raise Exception("Graph manifold missing torus")
    else:
        print "Graph manifold has fault"
```

⟨*faults.py*⟩≡
```
import regina
from sanity import *
```
⟨*trivially inessential*⟩
⟨*faults*⟩
⟨*how to find a fault the hard way*⟩
⟨*fault testing*⟩

4 Nonelementary embeddings

The above was a nice warmup. But for our problem, we don't need hyperbolic necklaces—we need necklaces that embed nonelementarily into hyperbolic manifolds. For Mom-n manifolds with $n \leq 4$, Gabai, Meyerhoff, and Milley were able to show in [**?**] that this is a trivial distinction, using normal surface theory (not Regina, but normal surface theory as such) and a notion of complexity due to Matveev. However, bead-number 7 structures are Mom-5s, and the distinction is nontrivial in general.

There is an algorithm that, given a triangulation T of a link-complement M, will return a finite list L of finite-volume hyperbolic 3-manifolds such that the hyperbolic 3-manifolds into which M embeds nonelementarily are exactly the hyperbolic Dehn fillings of elements of L. However, this algorithm is quite complicated, due to its dependence upon a S^3-link complement algorithm whose foundations were laid by R. Budney.

Thus, instead, we are going to implement a simpler algorithm NE that returns not L but a finite superset of L.

4.1 Review

Recall the following.

Definition 4.1. Suppose $\phi : (M, S) \hookrightarrow (N, T)$ is a proper embedding of M, with S, T torus components of ∂M, ∂N respectively. This embedding is *nonelementary* when the induced map $\phi_* : \pi_1(M) \to \pi_1(N)$ has nonabelian image.

What we have shown in the preceding work is not just that N of low cusp volume admits an embedding from a necklace manifold of bead number at most 7, but also that it admits a *nonelementary* such embedding. So the results of running NE on all the necklace manifolds of bead number at most 7 and concatenating these lists together will yield a finite list of Dehn parents for hyperbolic 3-manifolds of low cusp volume.

Lemma 4.2. *Suppose $\phi : (M, S) \hookrightarrow (N, T)$ is a nonelementary embedding of a link complement M in a hyperbolic link complement N.*

There is a nonelementary embedding $\phi' : (M, S) \hookrightarrow (N, T)$ with $\phi'(\pi_1(M)) = \phi(\pi_1(M))$ and such that for every boundary torus t of M, $\phi'(t)$ either is boundary parallel in N, or bounds a solid torus in N.

Definition 4.3. We call such an embedding a *Dehn embedding*.

In order to get a full nonelementary embedding recognition algorithm, one would need to refine the following lemma, part **??**, to account for *how* the two components get identified along their particular torus boundaries. This would start to involve a computation of the *companionship graph* of the JSJ-decomposition of M, an involved process (see [**?**]). We do not go into this here and content ourselves with using the lemma below as it stands.

Lemma 4.4. *Suppose $\phi : (M, S) \hookrightarrow (N, T)$ is a Dehn embedding of a link complement M in a hyperbolic link complement N.*

1. *M admits no $\mathbf{RP}^2$ or K^2.*
2. *Suppose M admits an essential sphere σ.*
   (a) *M is a connect-sum along σ.*
   (b) *Letting M' be that summand containing S, there is a Dehn embedding $\phi' : (M', S) \hookrightarrow (N, T)$.*
   (c) *The other summand is an S^3 link complement.*
3. *If M is irreducible, then it is ∂-irreducible.*
4. *Suppose M is irreducible and admits an essential torus τ.*
   (a) *τ separates M.*
   (b) *Letting M' be that component containing S and the other component be M'', one of the following is true:*
      i. *There is a Dehn embedding $\phi' : (M', S) \hookrightarrow (N, T)$, and M'' is the complement of a link in $D^2 \times S^1$.*
      ii. *There is a Dehn embedding $\phi' : (M'', \tau) \hookrightarrow (N, T)$, and M' is the complement of a link in $T^2 \times I$.*
5. *Suppose M is irreducible and atoroidal. Then M is hyperbolic.*

The proof comes more or less verbatim from the introductory lemmas of [**?**]. We leave the details to the reader.

4.2 Initial pseudocode sketch

Having finished the background review, we now sketch the structure of the routine `NE`, beginning with some naive ideas, and then refining them.

4.2.1 Naive pseudocode

Recall that NE should return a list of hyperbolic manifolds whose Dehn fillings include all hyperbolic link complements into which the given manifold embeds nonelementarily. Happily, we also already know the particular torus boundary component that should get sent to the small cusp, namely the polar cusp of a necklace manifold. So we can give that as input to the procedure, and implement it following the lemma without much thought.

```
NaiveNE(M,S):
    if there is P2 or K2 in M:
        return []
    else if there is nonseparating S2 or T2 in M:
        return []
    else if there is essential separating S2 ‘s’ in M:
        let M’, M’’ be components of M - s /
            with M’ that component containing S
        if M’’ not S3 link complement:
            return []
        else:
            return Naive(M’,S)
    else if M is D2 x S1:
        return []
    else if there is essential T2 ‘t’ in M:
        let M’, M’’ be components of M - t /
            with M’ that component containing S
        X,Y = [],[]
        if M’’ is D2 x S1 link complement:
            X = NaiveNE(M’,S)
        if M’ is T2 x I link complement:
            Y = NaiveNE(M’’,t)
        return union(X,Y)
    else if there is essential M2 or A2 in M:
        return []
    else:
        return [M]
```

4.2.2 Less naive pseudocode

Unfortunately we don't have a nice implementation of determining whether or not M'' is a certain kind of link complement. Instead, we will have some heuristic guess, which we hope will tell us whether or not a manifold is a given type of link complement. We call this being (or not being) an "obvious" link complement of the given type. If a manifold is obviously *not* such a manifold, then we can proceed as in the naive pseudocode. Otherwise we must give the manifold the benefit of the doubt—it might be such a link complement. We could say that we recur on NE when the other component is "not obviously not" a link complement of given type. But this awkward construction is captured neatly by `else` statements in the pseudocode.

```
NE(M,S):
    if there is P2 or K2 in M:
        return []
    else if there is nonseparating S2 or T2 in M:
        return []
    else if there is essential separating S2 ‘s’ in M:
        let M’, M’’ be components of M - s /
            with M’ that component containing S
        if M’’ is obviously not an S3 link complement
            return []
        else:
            return NE(M’,S)
    else if M is D2 x S1:
        return []
    else if there is essential T2 ‘t’ in M:
        let M’, M’’ be components of M - t /
            with M’ that component containing t
        X,Y = [],[]
        if M’’ is obviously not a D2 x S1 link complement:
            return []
        else:
            X = NE(M’, S)
        if M’ is obviously not a T2 x I link complement:
            return []
        else:
            Y = NE(M’’,t)
        return union(X,Y)
    else if there is essential M2 or A2 in M:
        return []
    else:
        return [M]
```

4.3 Implementing NE

With the sketch done, we move on to the actual implementation.

4.3.1 Obvious non-link-complements

First, we should define what it means to be obviously not a —— link complement with the blank filled variously by S^3, $D^2 \times S^1$, and $T^2 \times I$. We could just say that this is *never* obvious and get a correct procedure, but we can do better than that.

First of all, if M is closed and not S^3, then it is obviously not an S^3 link complement! Likewise, if it has at most one boundary torus and isn't $D^2 \times S^1$, then it is obviously not a $D^2 \times S^1$ link complement, and similarly if it has at most two boundary tori and isn't $T^2 \times I$, then it is obviously not a $T^2 \times I$ link complement.

Likewise, we should check that the homology of the manifold agrees appropriately with Alexander duality. ($D^2 \times S^1$- and $T^2 \times I$-link complements are themselves S^3-link complements.)

⟨*obvious link complement tests*⟩≡
```
  def couldBeLinkIn(M,amb):
      s3 = "S^3"
      d2s1 = "D^2 x S^1"
      t2i = "T^2 x I"
      try:
          assert amb in [s3, d2s1, t2i]
      except AssertionError:
          raise Exception("Only S^3, D^2 x S^1, and T^2 x I allowed as second argument")
      if not isLinkComplement(M,True):
          return False
```

After the above sanity check (really, a type-check), we do the boundary-component tests.

⟨*obvious link complement tests*⟩+≡
```
      nbc = M.countBoundaryComponents()
      if amb == s3:
          if nbc == 0:
              if not M.isThreeSphere():
                  return False
              else:
                  return True
      if amb == d2s1:
          if nbc == 0:
              return False
          if nbc == 1:
              if not M.isSolidTorus():
                  return False
              else:
                  return True
      if amb == t2i:
          if nbc in [0,1]:
              return False
          if nbc == 2:
              if not isT2xI(M):
                  return False
              else:
                  return True
```

Now having checked the boundary components are appropriate, we make sure the manifold's homology agrees with Alexander duality. Recall that by Alexander duality, if M is an S^3-link complement (which includes $D^2 \times S^1$- and $T^2 \times I$-link complements), then

$$H_1(M; \mathbf{Z}) = \mathbf{Z}^{|\pi_0(\partial M)|} \text{ and } H_2(M; \mathbf{Z}) = \mathbf{Z}^{|\pi_0(\partial M)|-1}.$$

⟨*obvious link complement tests*⟩+≡
```
h1 = M.homologyH1()
if h1.countInvariantFactors() > 0 \
   or h1.rank() != nbc:
   return False
h2 = M.homologyH2()
if h2.countInvariantFactors() > 0 \
   or h2.rank() != nbc-1:
   return False
```

If at this point we haven't determined whether or not the manifold is a link complement in whichever ambient manifold, then we give up. It could still be a link in the given manifold.

⟨*obvious link complement tests*⟩+≡
```
else:
    return True
```

4.3.2 NE

The code for NE is much the same as for the hyperbolicity algorithm, except that instead of terminating when we find a fault, we cut along the fault and examine the components recursively.

We also note that the following implementation is significantly different even from the above "less naive" pseudocode, as this implementation does not account for which boundary component of M is S. This could possibly include extraneous manifolds in the resulting list—but it is still a superset of Dehn parents as desired. (We don't account for S as Regina's surface cutting methods complicate the boundary. With more work one could keep track of which surface is which.)

We assume our input manifold is given as an ideal triangulation.

⟨*NE*⟩≡
```
def NE(ideal):
    Mi = regina.Triangulation3(ideal)
    Mi.intelligentSimplify()
```

Before any normal surface enumeration, we check to see if M's fundamental group is subtle. If it is not subtle, then either the group is not hyperbolic, or it is hyperbolic and recognisable (for instance, the figure-eight knot group is recognisable). As decided above, we deal with non-subtle presentations separately.

⟨*NE*⟩+≡
```
if regina.versionString() == '5.1':
    groups = set()
    x = hasCommonAxisObstruction(Mi)
    if x[0]:
        return []
```

Next, we do census checks on both the given triangulation, and its simplified idealization. (This also depends on using version 5.1, but that is already incorporated into `censusNE`.)

⟨*NE*⟩+≡
```
x = censusNE(Mi)
if x != None:
   if x[0]:
       return [Mi]
   if not x[0]:
       return []
```

Next, we check to see whether or not M admits a strict angle structure. If it does, then M is hyperbolic, and every hyperbolic N into which M embeds nonelementarily is a hyperbolic Dehn filling of M. This is a relatively quick check.

⟨*NE*⟩+≡
```
if Mi.hasStrictAngleStructure():
    return [Mi]
```

Having done our preliminary sanity checks, we now set up two normal surface collections: first, the fundamental-standard collection for an ideal triangulation of M, or the FSI collection; and the quad-vertex collection for a material triangulation of M (which Regina calls a "finite" triangulation). We call that the QVM collection. We can find all essential surfaces we need in the QVM collection; however, we suspect the FSI collection is smaller, and it contains all the essential *closed* surfaces that we need. So we set it up and look through it first, insofar as that is possible.

⟨*NE*⟩+≡
```
fsi = regina.NormalSurfaces.enumerate(Mi,regina.NS_STANDARD,regina.NS_FUNDAMENTAL)
```

Next, we look for obviously bad surfaces. Assuming M is orientable, we can lump these bad surfaces all together as closed non-separating surfaces of nonnegative Euler characteristic. (Remember, we can't look for surfaces with boundary using an ideal triangulation.)

⟨*NE*⟩+≡
```
assert ideal.isOrientable()
for i in range(fsi.size()):
    s = fsi.surface(i)
    x = s.eulerChar()
    if x >= 0 and s.cutAlong().isConnected():
        return []
```

Next, we look for an essential, connect-sum sphere. But we already know how to do this. Then we undo the connect-sum and test the summands for being link complements as in the above pseudocode. We undo a connect-sum by cutting along a sphere, then idealizing. (As above, this slow, obvious cutting code works just fine for us. But one ought to be able to improve it with crushing.)

⟨*NE*⟩+≡
```
s2 = essentialS2In(fsi)
if not s2 == None:
    s2 = fsi.surface(s2[0])
    MM = s2.cutAlong()
    MM.intelligentSimplify()
    MM.finiteToIdeal()
    MM.splitIntoComponents()
    mp = MM.firstChild()
    mpp = mp.nextSibling()
    Xp = couldBeLinkIn(mp,"S^3")
    Xpp = couldBeLinkIn(mpp,"S^3")
    Lp,Lpp = [],[]
    if Xp:
        Lpp = NE(mpp)
    if Xpp:
        Lp = NE(mp)
    return Lp + Lpp
```

We follow the reducibility test by a boundary irreducibility test. If we finish the above S^2 search with no essential S^2, then M is irreducible. Since M is assumed to be a non-trivial link complement, it is therefore ∂-irreducible if and only if it is a solid torus. So we don't need to do a disc search ourselves, and can leave this to Regina. We can put off the quad-normal surface enumeration till the very end.

Of course, $D^2 \times S^1$ has fundamental group $\mathbf{Z}$, so it has no nonelementary embeddings into any manifold whatever.

⟨*NE*⟩+≡
```
if ideal.isSolidTorus():
    return []
```

Next, we look for an essential separating T^2. We already found no non-separating T^2. So all the tori in our list are separating. It remains to find an essential torus among these. But we already know how to do this. If we find such a torus, then we cut along it, and test the components for being link-complements. This implementation differs from the above pseudocode in that no $T^2 \times I$-link complement test is performed. That is because we cannot yet keep track of boundary components in Regina under cutting.

⟨*NE*⟩+≡
```
t2 = essentialT2In(fsi)
if t2 != None:
    t2 = fsi.surface(t2[0])
    MM = t2.cutAlong()
    MM.intelligentSimplify()
    MM.finiteToIdeal()
    MM.splitIntoComponents()
    mp = MM.firstChild()
    mpp = mp.nextSibling()
    Xp = couldBeLinkIn(mp,"D^2 x S^1")
    Xpp = couldBeLinkIn(mpp,"D^2 x S^1")
    Lp,Lpp = [],[]
    if Xp != False:
        Lpp = NE(mpp)
    if Xpp != False:
        Lp = NE(mp)
    return Lp + Lpp
```

Finally, if there are no obviously bad surfaces, nor essential separating S^2s nor T^2s nor essential discs, then M is irreducible, ∂-irreducible, and (geometrically) atoroidal. Thus M is either hyperbolic or atoroidal Seifert-fibered. We can distinguish these two cases, with our assumptions on M, using surfaces: it is Seifert-fibered if and only if it admits an essential annulus or Möbius band. By Q-normal surface theory we can always find such a surface among quad-vertex surfaces, under our assumptions on M.

Most importantly, a Seifert-fibered manifold admits no nonelementary embeddings into a hyperbolic link complement. So we don't need any recursive calls to `NE` in this case.

⟨*NE*⟩+≡
```
Mf = regina.Triangulation3(Mi)
Mf.idealToFinite()
Mf.intelligentSimplify()
qvm = regina.NormalSurfaces.enumerate(Mf,regina.NS_QUAD,regina.NS_VERTEX)
m2 = essentialM2In(qvm)
if m2 != None:
    return []
a2 = essentialA2In(qvm)
if a2 != None:
    return []
```

If we have found no faults, then M is hyperbolic.

⟨*NE*⟩+≡
```
return [ideal]
```

That concludes the procedure `NE`.

4.4 Testing

Our test cases are a connect-sum of two figure-eight knot exteriors, and two Whitehead links glued along two of their boundaries.

The first test case should state that the connect-sum of two figure-eight knot exteriors embeds nonelementarily into a hyperbolic knot complement.

The second test case should state that two Whitehead links, glued along one boundary torus each, splits up along an essential torus into two Whitehead link exteriors, recognizable via a census check, and the hyperbolic knot complements into which the glued-up manifold embeds nonelementarily are included in the hyperbolic knot complements into which each Whitehead link exteriors embed nonelementarily.

⟨*nonelementary testing*⟩≡
```
if __name__ == "__main__":
    f82 = regina.Triangulation3('kLLAAzMkceeeffhhjjjadlqaaaamhl')
    whl2 = regina.Triangulation3('mLALLLMPQbcceghjkjllklmlbsakapsoknf')
    print str(NE(f82))
    print str(NE(whl2))
```

⟨*nonElementary.py*⟩≡
```
import regina
import snappy
from faults import *
from sanity import *
⟨obvious link complement tests⟩
⟨NE⟩
⟨nonelementary testing⟩
```

5 Exceptional fillings

Our final crucial computation is to determine those one-cusped hyperbolic Dehn fillings N of some collection of two-cusped hyperbolic 3-manifolds that have more than eight exceptional slopes. We first determine a finite collection of one-cusped Dehn fillings that might have more than eight exceptional slopes. Then we conclude by determining which among these do in fact have more than 8 exceptional slopes.

Here is our plan in more detail. Fix M. If s is a slope on a boundary torus t of M with length greater than 6, then $M(s)$ is hyperbolic by the 6 Theorem. The exceptional slopes s' of $M(s)$ we may identify with exceptional Dehn filling coefficients $s + s'$ on M. There are at most as many of these as there are short slopes s' on the other boundary torus t' of M.

So, first we will examine how many "short" slopes there are on each boundary torus. For every two-cusped manifold we inspect and for each of its boundary tori, this number is always at most 8. The only three-cusped manifold we encounter is the magic manifold `s776`, whose exceptional fillings were already classified in [**?**]. Therefore, the candidate one-cusped manifolds are the hyperbolic Dehn fillings on our two-cusped manifolds along short slopes.

Finally, for each candidate one-cusped manifold, we determine its short slopes, and determine an upper bound on how many of these are exceptional slopes. This upper bound is always at most 8, except of course for the figure eight knot complement `m004`, which has 10 exceptional slopes.

5.1 Enumerating short slopes

We now describe a simple, efficient approach to enumerating short slopes inspired by the first chapter of [**?**], where the reader can find a more geometric intuition for the following. A different approach was taken in [**?**]. The following code is simpler and goes through fewer slopes. We also prove this code in more detail than was given for the code in [**?**]. The reader comfortable with [**?**] could probably skip directly to the code and understand its correctness without much difficulty, as a reader comfortable with the properties of quadratic forms used there could comfortably understand its correctness with the justifications given in that paper.

5.1.1 Quadratic forms on cusps

Given a maximal cusp C in some hyperbolic 3-manifold M, we can have SnapPea calculate the length ℓ of the shortest nontrivial translation m on the cusp, and calculate the shape z of the cusp. Identifying $H_1(C;\mathbf{R}) = \mathbf{C}$ by the orientation-preserving isometry φ taking m to ℓ, we can calculate a second-shortest translation $n = \varphi^{-1}(z \cdot \ell)$. Its length is $|z| \cdot \ell$. Finally, we calculate that the length of $p \cdot m + q \cdot n$ is $|p \cdot \ell + q \cdot \ell \cdot z|^2 = \ell^2 \cdot |p + q \cdot z|^2$. Thus we get a positive-definite real quadratic form $Q(p,q) = \ell^2 \cdot |p + q \cdot z|^2$, taking its smallest and second-smallest values at $(1,0)$ and $(0,1)$ respectively.

Lemma 5.1. *Suppose $Q(v) \leq Q(w) \leq Q(v+w)$ and $1 \leq p, 1 \leq q$ with $p, q \in \mathbf{N}$. Then $Q(p \cdot v + q \cdot w) \geq Q(v+w)$.*

Proof.

$$\begin{aligned}
Q(v+w) &= Q(v) + Q(w) + 2 \cdot \langle v, w \rangle \\
Q(v+w) - Q(w) &= Q(v) + 2 \cdot \langle v, w \rangle \geq 0 \\
\langle v, w \rangle &\geq -Q(v)/2 \\
Q(p \cdot v + q \cdot w) &= p^2 \cdot Q(v) + q^2 \cdot Q(w) + 2 \cdot p \cdot q \cdot \langle v, w \rangle \\
Q(p \cdot v + q \cdot w) - Q(v+w) &= (p^2 - 1) \cdot Q(v) + (q^2 - 1) \cdot Q(w) + 2 \cdot (p \cdot q - 1) \cdot \langle v, w \rangle \\
&\geq (p^2 - 1) \cdot Q(v) + (q^2 - 1) \cdot Q(w) - (p \cdot q - 1) \cdot Q(v) \\
&= (p^2 - p \cdot q) \cdot Q(v) + (q^2 - 1) \cdot Q(w) \\
&\geq (p^2 + q^2 - p \cdot q - 1) \cdot Q(v)
\end{aligned}$$

(We used the conditions on p, q to show $2 \cdot (p \cdot q - 1) \cdot \langle v, w \rangle \geq -(p \cdot q - 1) \cdot Q(v)$.) Now, $q(p,q) = p^2 - p \cdot q + q^2$ is itself a positive-definite binary quadratic form whose smallest value on $\mathbf{Z}^2 \setminus \{\vec{0}\}$ is 1. The result follows. □

5.1.2 The tree in the background

From the above lemma we get a way to show that infinitely many slopes are long. However, the preconditions of the lemma do not use a single vector, but two vectors. We therefore find it useful to put some structure not on $\mathbf{Z}^2$ but on (a certain subset of) $M_{2\times 2}\mathbf{Z}$.

Let T be the subset of $M_{2\times 2}\mathbf{N}$ consisting of determinant one matrices. Suppose $M \in T$. Let

$$M = \begin{pmatrix} a & b \\ c & d \end{pmatrix}.$$

Then $a \cdot d - b \cdot c = 1$, so $d/b = c/a + 1/(ab) \geq c/a$, assuming $a \cdot b > 0$. On the other hand, if $a = 0$, then $-b \cdot c = 1$, which is impossible; and if $b = 0$, then $a \cdot d = 1$, in which case $a = d = 1$ and $d/b = \infty$. Regarding $\infty > x$ for all $x \in \mathbf{Q}$, in any case, $d/b > c/a$.

Definition 5.2. The *interval of* M is the open interval $I(M) = (c/a, d/b)$.

Furthermore, again since $a \cdot d - b \cdot c = 1$, $c/a < (c+d)/(a+b) < d/b$. The *left child of* M is the matrix

$$L(M) = \begin{pmatrix} a & a+b \\ c & c+d \end{pmatrix} = M.\begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix},$$

and the *right child of* M is the matrix

$$R(M) = \begin{pmatrix} a+b & b \\ c+d & d \end{pmatrix} = M.\begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}.$$

The *slope* of M is $s(M) = (a+b \;\; c+d)^t = M.(1\ 1)^t$.

The *number* of M is $r(M) = (c+d)/(a+b)$.

We see that $L(M), R(M) \in T$ if $M \in T$. By the above considerations, $I(L(M)) \cap I(R(M)) = \emptyset$ and $I(L(M)) \cup \{r(M)\} \cup I(R(M)) = I(M)$. In particular, $I(L(M))$ and $I(R(M))$ are strict subsets of $I(M)$, and $r(M) \in I(M)$.

Clearly we may also define the inverse maps

$$L^{-1}(M) = \begin{pmatrix} a & b-a \\ c & d-c \end{pmatrix}$$

and

$$R^{-1}(M) = \begin{pmatrix} a-b & b \\ c-d & d \end{pmatrix}$$

as maps from $SL_2\mathbf{Z}$ to itself; however, neither inverse map takes T into itself.

Definition 5.3. By abuse of notation, we will regard T as a digraph that has its elements for vertices and that has edges from each vertex to its left child and right child.

Lemma 5.4. *T is an infinite binary tree whose source (i.e. root) is the identity matrix.*

Proof. Note that the interval of the identity matrix is $(0, \infty)$, and every nonidentity matrix has interval being a strict subset of this interval. So the identity is a child of no vertex—that is, it is a source.

Conversely, suppose v is a child of no vertex. Let

$$v = \begin{pmatrix} a & b \\ c & d \end{pmatrix}.$$

Then neither $L^{-1}(v)$ nor $R^{-1}(v)$ lies in T. Thus one of $b-a, d-c$ is negative, and one of $a-b, c-d$ is negative—i.e., $a-b, c-d$ are of opposite sign. If $a-b>0$, then $a>b$ and $d>c$. So $a \geq b+1$ and $d \geq c+1$; hence $a \cdot d \geq b \cdot c + b + c + 1$, and $a \cdot d - b \cdot c = 1 \geq b + c + 1$, i. e. $0 \geq b + c$. So $b = c = 0$ and $a = d = 1$. The case $a - b < 0$ leads similarly to $a \cdot d - b \cdot c = 1 \leq -a - d - 1$, which is impossible. So the identity is the unique source of this digraph.

On the other hand, suppose v is a child of some vertex w. If $v = L(w)$, then $v_{0,0} < v_{0,1}$ and $v_{1,0} < v_{1,1}$, so $R^{-1}(v) \notin T$, and thus v is not the right child of any vertex. So v is the child of the unique vertex $L^{-1}(v) = w$. A similar result holds if $v = R(w)$. Thus vertices that are children have unique "parents." That is, each non-identity vertex has a unique in-edge. Each vertex also has exactly two out-edges. Thus T is 3-regular apart from the source.

Finally, every vertex has a unique backtrack-free path from the source: simply apply L^{-1} and R^{-1} repeatedly until one can do so no longer. This terminates since both maps strictly reduce L_1 norm on T, and L_1 norm takes values in $\mathbf{N}$ on T. So T is a tree.

Therefore, T is an infinite binary search tree. □

Remark 5.5. We never actually construct this tree T or even any finite subtree of it in code. Instead we construct a function whose call-structure is equivalent to finite subtrees of this tree. We use T to prove properties of the function.

Definition 5.6. The *descendant relation* is the reflexive transitive closure of the child relation. That is, "is a descendant of" is the smallest relation such that N is a descendant of M if and only if either $N = M$ or N is a child of a descendant of M. We write $M \to N$ for "N is a descendant of M".

Lemma 5.7. *All rows and columns of elements of T are primitive. (In particular, all slopes of elements are primitive.)*

Proof. Suppose $M \in T$; let

$$M = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$$

. Suppose, say, $(a\ c) = k \cdot (A\ C)$. Then

$$1 = \det M = k \cdot \begin{vmatrix} A & b \\ C & d \end{vmatrix}.$$

Hence $k = \pm 1$, and $(a\ c)$ is primitive. The result follows from symmetries of det. □

Lemma 5.8. *Every $(p\ q)^t \in \mathbf{N}^2$ with $gcd(p, q) = 1$ is the slope of a unique element of T.*

Proof. We give an algorithm in Python that computes this element. (But we don't use this algorithm in our computation.)

⟨*T element from slope*⟩≡
```
def T_element(slope):
    (p,q) = slope
    u,v,a,b,c,d = p,q,1,0,0,1
    while u != v:
        if u > v:
           Q,R = u / v, u % v
           u,a,b = R, a + Q*c, b + Q*d
        elif v > u:
           Q,R = v / u, v % u
           v,c,d = R, c + Q*a, d + Q*b
    return (a,b,c,d)
```

First, none of the code modifies p or q.

Before the loop, we have set up the preconditions $u = a \cdot p - b \cdot q$, $v = -c \cdot p + d \cdot q$, and $a \cdot d - b \cdot c = 1$. The loop keeps these conditions invariant. For instance, if $u > v$, then we calculate Q, R such that $0 \leq R < v$ and $u = Q \cdot v + R$, then reassign $u := R$, i.e. $u := u - Q \cdot v$. Now,

$$\begin{aligned} u - Q \cdot v &= a \cdot p - b \cdot q - (-Q \cdot c \cdot p + Q \cdot d \cdot q) \\ &= a \cdot p - b \cdot q + Q \cdot c \cdot p - Q \cdot d \cdot q \\ &= (a + Q \cdot c) \cdot p - (b + Q \cdot d) \cdot q \end{aligned}$$

Moreover, the substitution $a, c := a + Q \cdot b, c + Q \cdot d$ does not change the determinant $a \cdot d - b \cdot c = 1$. The other cases are similar.

Furthermore, the loop has $max(u, v)$ as a variant, i.e. as a natural number that decreases with every loop. The reason for this is that $u, v \geq 1$; this follows since $gcd(u, v) = gcd(p, q) = 1$. So the loop terminates.

As usual, it terminates in the condition that is the conjunction of the above preconditions together with the negation $u = v$ of the loop condition. In this case, $u = v = gcd(u, v) = gcd(p, q) = 1$. Now $a \cdot p - b \cdot q = 1 = -c \cdot p + d \cdot q$ implies $(c + d)/(a + b) = q/p$. Furthermore, the left- and right-hand sides of this equation are in reduced terms, since $gcd(p, q) = 1$ and $a \cdot d - b \cdot c = 1$. Since their numerators and denominators are at least 0, they must have equal numerators and denominators. This shows the existence of the element.

Uniqueness follows from considering intervals. Suppose $v = (p\ q)^t$ is the slope of M and M'. Then the number of v is in both $I(M)$ and $I(M')$. So $I(M) \cap I(M')$ is nonempty. Furthermore, no strict subintervals (of the form $I(Y)$ for some $Y \in T$) of $I(M)$ or $I(M')$ contain v. So $I(M) \cap I(M') = I(M) = I(M')$. That is, $M = M'$. □

Lemma 5.9. *As a monoid, T acts on the left of its associated digraph by monomorphisms.*

Proof. We just need to check that it acts by injections that preserve the child relations. Of course multiplication acts by injections. Suppose $X, M \in T$. We wish to show $L(X.M) = X.L(M)$ and $R(X.M) = X.R(M)$, so left-multiplication preserves the child relations. But this is trivial by associativity, since there are matrices ℓ, r such that for all matrices s, $L(s) = s.\ell$ and $R(s) = s.r$ (see the definition of L and R). □

Lemma 5.10. *Suppose $M \in T$ and v, w are the columns of M. The descendants of M are precisely those $N \in T$ whose slopes are of the form $p \cdot v + q \cdot w$ with $1 \leq p$, $1 \leq q$.*

Proof. Suppose N is a descendant of M. By induction, the slope of N is of the form $p \cdot v + q \cdot w$ with $1 \leq p$, $1 \leq q$, $p, q \in \mathbf{N}$.

Conversely, suppose the slope of N is of that form $p \cdot v + q \cdot w$ with $1 \leq p$, $1 \leq q$. Then, first of all, $v = (p\ q)^t$ is a primitive; otherwise, the slope of N would not be a primitive. Hence $gcd(p, q) = 1$. Thus it is itself the slope of some $X \in T$. Let $k = (1\ 1)^t$; then for all matrices Y, $s(Y) = Y.k$. So $v = s(X) = X.k$; hence $s(N) = M.v = M.X.k = s(M.X)$. So $N = M.X$ by uniqueness. Since T acts by monomorphisms, and since X is a descendant of the identity, N is a descendant of M. □

5.1.3 Short slope enumeration

Proposition 5.11. *Suppose $M \in T$, and suppose Q is a positive-definite quadratic form on $\mathbf{Z}^2$ taking smallest and second-smallest values on $(1\ 0)^t$ and $(0\ 1)^t$, respectively.*

If N is a descendant of M, then $Q(s(N)) > Q(s(M))$.

Proof. This follows directly from lemmas **??** and **??**. □

Theorem 5.12. *Assuming f does not change any symbols used in the following program (e.g. M, f, `forAllShortDescendants`, `+`, and so on), and assuming Q takes values in a decidable field, the following program does $f(N)$ for all descendants N of M such that $Q(s(N)) \leq B$, and for no other descendants of M.*

⟨*shortSlopes.py*⟩≡

```
def forAllShortDescendants(Q,B,M,f):
    (v, w) = M
    S = (v[0]+w[0],v[1]+w[1])
    if Q(S) > B:
        pass
    else:
        forAllShortDescendants(Q,B,(v,S),f)
        f(M)
        forAllShortDescendants(Q,B,(S,w),f)
```

Proof. Let D be the subgraph of T whose vertices are the descendants of M whose slopes have Q-value at most B (and whose edges are the edges among these vertices in T). We will show D is a finite binary search tree on the Q-values of the slopes.

Suppose N is a descendant of M in D with $Q(s(N)) < B$. Suppose $p : [0, n] \to T$ is a backtrack-free path in T from $p(0) = M$ to $p(n) = N$. Then by Proposition **??**, $Q(s(p(i))) < B$ for all $0 \leq i \leq n$. So p is a backtrack-free path in D between M and N. Hence N is a descendant of M in D. Thus D is a sub-tree of T.

To show that D is finite, it suffices to show it has finitely many vertices. Its vertices are, by Lemma **??**, identified with the slopes in $\mathbf{N}^2$ lying in the set $E = \{v : \mathbf{R}^2 \mid Q(v) \leq B\}$, an ellipse. In particular, E is compact; since $\mathbf{N}^2$ is discrete in $\mathbf{R}^2$, D is finite.

The result follows by structural induction on D. □

Corollary 5.13. *Assuming f changes none of the symbols of the following program, and assuming Q is a positive-definite binary quadratic form taking values in a decidable field, the following program does $f(v)$ for all primitives $v \in \mathbf{Z}^2$ in the right half-plane, and possibly also for $v = (0, 1)$, such that $Q(v) \leq B$.*

⟨*shortSlopes.py*⟩+≡
```
def forAllShortSlopes(Q,B,f):
    s = lambda M : (M[0][0]+M[1][0],M[0][1]+M[1][1])
    g = lambda M : f(s(M))
    Qbar = lambda (p,q): Q((p,-q))
    Lambda = lambda ((a,c),(b,d)): ((b,-d),(a,-c))
    gbar = lambda M : g(Lambda(M))
    forAllShortDescendants(Qbar,B,((0,1),(1,0)),gbar)
    if not Q((1,0)) > B:
        f((1,0))
    forAllShortDescendants(Q,B,((1,0),(0,1)),g)
    if not Q((0,1)) > B:
        f((0,1))
```

Proof. Let us abbreviate the name of the program in Theorem **??** to `fasd`. By that theorem, $fasd(Q, B, I, g)$ does $g(M) = f(s(M))$ for all descendants M of I with $Q(s(M)) \leq B$, which is to say it does $f(v)$ for all primitives $v \in (\mathbf{N} + 1)^2$ with $Q(v) \leq B$. We also do $f((1, 0))$ and $f((0, 1))$ if need be.

It remains to show $fasd(\bar{Q}, B, I, \bar{g})$ does $f(v)$ for all primitives $v \in (\mathbf{N} + 1) \times (-\mathbf{N} - 1)$. Let $\Lambda(M) = \sigma.M.\tau$ where

$$\sigma = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}, \qquad \tau = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}.$$

We have defined $\bar{Q}((p, q)) = Q((p, -q))$ and $\bar{g}(M) = g(\Lambda(M))$. Then by Theorem **??**, $fasd(\bar{Q}, B, I, \bar{g})$ does $\bar{g}(M)$ for all descendants M of I with $\bar{Q}(s(M)) \leq B$. Suppose $v = (p, -q)$ with $1 \leq p, q \in \mathbf{N}$. Suppose further that $Q(v) \leq B$. Let M be the unique descendant of I with slope (p, q). Then $\bar{Q}(s(M)) = Q(v)$, and $\bar{g}(M) = g(\Lambda(M)) = f(s(\Lambda(M))) = f(v)$. The result follows. □

5.1.4 Extracting a quadratic form

It remains only to extract a positive-definite rational lower bound R on the Riemannian form restricted to a maximal cusp of a given hyperbolic 3-manifold M. Using `SnapPy` as a Sage module, this is quite easily done.

⟨*cuspForm.py*⟩≡
```
def cuspFormLowerBound(mfld,cusp):
    X = mfld.cusp_translations(verified=True)
    (z,w) = X[cusp]
    nrm = lambda Z: (Z*Z.conjugate()).real()
    q = lambda (p,q): nrm(p*z+q*w).lower()
    return q
```

This gives us a function that is a lower bound on the actual quadratic form. To guarantee termination for the algorithm, it would be best to get a lower bound that is a rational positive-definite quadratic form. However, our procedure as defined above terminates with our desired input, so we don't worry about this.

We can now conclude with our computation.

6 The main computation

We have constructed primitives that we think should make it possible, or even simple, to implement an iterative, "fused" computation that investigates one gluing at a time, memoizing computations for later. I still think this is a worthwhile endeavor, but its implementation seems to be about a year or so beyond my current programming capabilities. So, as alluded to repeatedly above, we will take the following more naive approach to the computation. It splits into two parts: the nonelementary candidate enumeration, and the determination of exceptional fillings.

⟨*main preamble*⟩≡
```
import regina
import snappy
from necklace import *
from enumerateGluings import *
from sanity import *
from faults import *
from nonElementary import *
from DehnFillHack import *
from cuspForm import *
from shortSlopes import *
```

6.1 Nonelementary candidate enumeration

Before finally implementing everything let us review our goal and our plan to achieve that goal. Our goal for this first part of the computation is to enumerate a finite set $\mathcal{S}$ of hyperbolic 3-manifolds such that every hyperbolic knot complement into which a necklace gluing embeds nonelementarily is a Dehn filling of some element of $\mathcal{S}$. Our plan to achieve this goal is as follows:

1. Initialize an empty set `S`.
2. Enumerate the necklace gluings of bead number $n \leq 7$ and put their isomorphism signatures into a Python set `necklSigs`.
3. Construct a new set `mfldSigs` by constructing the associated triangulation of each signature in `necklSigs`, *simplifying that triangulation using Regina*, then putting the resulting signature into `mfldSigs`.
4. Filter `mfldSigs` via sanity checks to get a new set `saneSigs`.
5. For every `sigma` in `saneSigs`, call `NE(sigma)` and put the isomorphism signature of each element of `NE(sigma)` in `S`.
6. `S` represents the desired set $\mathcal{S}$.

Before diving into the above plan immediately, we note that we will memoize the arduous calls to `NE`, since it is so easy to do so with a global memoization table `NEmem`.

⟨*memoizing NE*⟩≡
```
memoTabNE = {}
def memoNE(ideal):
    sig = ideal.isoSig()
    if sig in memoTabNE:
        return memoTabNE[sig]
    x = NE(ideal)
    memoTabNE[sig] = x
    return x
```

Now we implement the above plan. The following code runs in about a minute on a 2009 iMac.

⟨*nonelementary candidate enumeration*⟩≡
```
S = set()
necklSigs = set()
for i in [4,5,6,7]:
    print "Enumerate necklace sigs of bead number " + str(i)
    necklSigs = necklSigs.union(necklaceIsoSigs(i))
mfldSigs = set()
print "Simplifying the necklace sigs..."
for sig in necklSigs:
    x = regina.Triangulation3(sig)
    x.intelligentSimplify()
    mfldSigs.add(x.isoSig())
saneSigs = set()
print "Performing sanity checks..."
for sig in mfldSigs:
    x = regina.Triangulation3(sig)
    if isLinkComplement(x,False):
        saneSigs.add(sig)
print "Now for the real work..."
for sig in saneSigs:
    print "Working on signature " + sig
    x = regina.Triangulation3(sig)
    parents = memoNE(x)
    for y in parents:
        S.add(y.isoSig())
```

We will have to treat three-cusped parents separately. Fortunately, there only ends up being one of those, namely `s776`. Upon the conclusion of this procedure, `S` has no three-cusped signatures.

⟨*three cusped parent determination*⟩≡
```
print "Here are the three-cusped parents."
c3 = set()
for sig in S:
    parent = regina.Triangulation3(sig)
    if parent.countBoundaryComponents() == 3:
        c3.add(sig)
        x = regina.Census.lookup(parent)
        c = x.first()
        if c == None:
            print sig + " is not in the census."
        else:
            print sig + " is '" + c.name() + "' in the census."
for sig in c3:
    S.remove(sig)
```

6.2 Dehn filling hack

We will want to fill our manifolds along slopes and determine whether or not the results are hyperbolic. Unfortunately SnapPy's Dehn filling triangulation routines are not deterministic, which can lead to large variance in running times. So we define the following technical routine for trying to Dehn fill in a consistent way. The `fuel` input is just a number indicating how many times we want to redo the Dehn filling.

⟨*Dehn filling hack*⟩≡
```
def minFill(snappyMfld,fuel):
    N = snappyMfld.filled_triangulation()
    x = N.triangulation_isosig(decorated=False)
    minimum = (N.num_tetrahedra(), x)
    for i in range(fuel):
        N = snappyMfld.filled_triangulation()
        x = N.triangulation_isosig(decorated=False)
        new = (N.num_tetrahedra(), x)
        if new < minimum:
            minimum = new
    return regina.Triangulation3(minimum[1])
```

6.3 Determining exceptional fillings

Having gotten our set $\mathcal{S}$ we now determine which hyperbolic knot complement Dehn fillings of elements of $\mathcal{S}$ admit more than eight exceptional Dehn fillings. Throughout this portion of the code we end up doing faultlessness testing; this is again onerous, especially after doing Dehn fillings, so we make sure to memoize our computations.

⟨*faultlessness memoization*⟩≡
```
faultMemo = {}
def memoIsFaultless(mfld):
    sig = mfld.isoSig()
    if sig in faultMemo:
        return faultMemo[sig]
    x = isFaultless(mfld)
    faultMemo[sig] = x
    return x
```

6.3.1 Checking a single filling

First, we will want to determine whether or not a slope on a cusp of a `snappy.Manifold` is exceptional or not. We make a copy `M` of the manifold, so as not to disturb the manifold itself. We then clear any Dehn fillings `M` might have, then fill.

⟨*filling type*⟩≡
```
  def fillingType(snappyMfld,cusp,slope):
      M = snappyMfld.copy()
      for cusp in range(M.num_cusps()):
          M.dehn_fill((0,0),cusp)
      M.dehn_fill(slope,cusp)
```

Now we try to get Snappy in Sage to prove that `M` is hyperbolic.

⟨*filling type*⟩+≡
```
      if M.verify_hyperbolicity()[0]:
          return ("Hyperbolic", "Immediate verification")
```

One is tempted to give up here and start triangulating the Dehn filling. But we will try harder to verify hyperbolicity if the volume is large enough. We drill out along curves in the dual graph and try verifying associated triangulations, canonized and not canonized.

⟨*filling type*⟩+≡
```
      try:
          if M.volume() > 0.9:
              sig = M.triangulation_isosig(decorated=False)
              for curve in M.dual_curves():
                  for tryCanonize in [True, False]:
                      A = M.drill(curve)
                      if tryCanonize:
                          try:
                              A.canonize()
                          except:
                              pass
                      A.dehn_fill((1,0),M.num_cusps())
                      if tryCanonize:
                          try:
                              A.canonize()
                          except:
                              pass
                      if A.verify_hyperbolicity()[0]:
                          reason = "Drill " + str(curve)[0] + " in " + sig
                          return ("Hyperbolic", reason)
      except ValueError:
          pass
```

If that does not prove hyperbolicity, then we try to disprove its hyperbolicity. We now triangulate the Dehn filling. Retriangulating ten times seems to be enough to get triangulations of consistently low size, if not isomorphic triangulations.

⟨*filling type*⟩+≡
```
      fuel = 10
      N = minFill(M,fuel)
```

Otherwise, after all this preliminary checking, we finally run the faultlessness routine in Regina. If the return value has first entry `False` then the manifold is certainly not hyperbolic, for the reason that is in the second entry.

⟨*filling type*⟩+≡
```
x = memoIsFaultless(N)
if x[0] == False:
    result = ("Not hyperbolic", x[1])
```

Otherwise, all we know is that N is faultless. If it's closed then it could possibly be small Seifert fibered, in which case the filling would be exceptional.

⟨*filling type*⟩+≡
```
if N.countBoundaryComponents() > 0:
    result = ("Hyperbolic", x[1])
else:
    result = ("Faultless", x[1])
return result
```

That concludes our method for checking whether or not a filling of a cusp of a manifold is exceptional.

6.3.2 Classifying short slopes

At several points in our procedure we will need to classify the short slopes of a cusp of a hyperbolic link complement. So we implement that here, using a short local function definition to fit it into the `forAllShortSlopes` framework.

⟨*classify short slopes*⟩≡
```
def classifyShortSlopes(snappyManifold, cusp):
    mu = snappyManifold.copy()
    q = cuspFormLowerBound(mu,cusp)
    hyp,nonhyp,unknown = [],[],[]
    def checkSlope(s):
        t = fillingType(mu,cusp,s)
        if t[0] == "Hyperbolic":
            hyp.append(s)
        if t[0] == "Not hyperbolic":
            nonhyp.append(s)
        if t[0] == "Faultless":
            unknown.append(s)
    forAllShortSlopes(q,36,checkSlope)
    return (hyp,nonhyp,unknown)
```

6.3.3 Determining the candidate knot complements

The next order of business is to determine the hyperbolic Dehn fillings of the elements of $\mathcal{S}$ that could possibly have more than eight exceptional Dehn fillings. We first show that it suffices to restrict our attention to short hyperbolic Dehn fillings, as there are never more than eight short fillings in the cusps of elements of $\mathcal{S}$.

⟨*bounding the number of short slopes on parents*⟩≡

```
print "Now we bound the number of short slopes."
for sig in S:
    print "Working on " + sig + "..."
    parent = snappy.Manifold(sig)
    if parent.num_cusps() != 2:
        print "Encountered " + parent.identify()[0].name() + " with " + str(parent.num_cusps()) + " cusps"
        continue
    for cusp in [0,1]:
        (hyp, nonhyp, unknown) = classifyShortSlopes(parent, cusp)
        if len(nonhyp) + len(unknown) > 8:
            print "Too many potentially exceptional slopes on cusp " + str(cusp) + " of " + sig
            break
    else:
        print "The number of possibly exceptional slopes was small enough."
```

Incidentally, the above also shows that the only one-cusped "parent" manifold in $\mathcal{S}$ is the figure-eight knot complement `m004`. So we've shown that $\mathcal{S}$ contains one one-cusped manifold `m004`, one three-cusped manifold `s776`, and, besides these, only two-cusped manifolds. The fillings of `s776` have been classified in [**?**]. Of course, `m004` has 10 exceptional fillings as described in [**?**]. It remains for us to investigate the two-cusped parents.

We next determine those one-cusped hyperbolic fillings of two-cusped elements of $\mathcal{S}$ along short slopes; one-cusped fillings along long slopes, by the above calculation, can have no more than 8 exceptional slopes.

⟨*determining one-cusped candidates*⟩≡

```
c1names = set()
c1sigs = set()
for sig in S:
    parent = snappy.Manifold(sig)
    parent.canonize()
    print "Working on " + parent.identify()[0].name() + "..."
    if parent.num_cusps() != 2:
        continue
    for cusp in [0,1]:
        print "Working on cusp " + str(cusp)
        parent.dehn_fill((0,0),0)
        parent.dehn_fill((0,0),1)
        (hyp,nonhyp,unknown) = classifyShortSlopes(parent,cusp)
        assert unknown == []
        for slope in hyp:
            print "Working on slope " + str(slope) + "..."
            parent.dehn_fill(slope,cusp)
            N = parent.filled_triangulation()
            x = N.triangulation_isosig(decorated=False)
            n = regina.Triangulation3(x)
            n.intelligentSimplify()
            n.idealToFinite()
            n.intelligentSimplify()
            n.finiteToIdeal()
            n.intelligentSimplify()
            try:
                N.simplify()
                N.canonize()
                c1names.add(N.identify()[0].name())
            except:
                c1sigs.add(n.isoSig())
for name in c1names:
    print name
for sig in c1sigs:
    print sig
```

6.4 Determining exceptional fillings

Let's say a hyperbolic knot complement with at least nine exceptional fillings is a *superexceptional* knot complement. We have now constructed a superset `c1` of the set of superexceptional knot complements into which necklace gluings embed nonelementarily. Now we determine which among them are (in fact, is) superexceptional. (We add `m004` back into `c1` at this point, having finished our investigation of the two-cusped manifolds.)

⟨*determining exceptional fillings*⟩≡
```
superexceptional = []
for sig in c1sigs:
    print "Working on " + sig + "..."

    m = snappy.Manifold(sig)
    (hyp,nonhyp,unknown) = classifyShortSlopes(m,0)
    if len(nonhyp) + len(unknown) <= 8:
        print sig + " is not superexceptional."
    else:
        print sig + " is superexceptional."
        superexceptional.append(sig)
print "The superexceptionals are as follows: " + str(superexceptional) + "."
```

⟨*chubby.py*⟩≡
```
⟨main preamble⟩
⟨memoizing NE⟩
⟨faultlessness memoization⟩
⟨filling type⟩
⟨classify short slopes⟩
⟨Dehn filling hack⟩
if __name__ == "__main__":
    ⟨nonelementary candidate enumeration⟩
    ⟨three cusped parent determination⟩
    ⟨bounding the number of short slopes on parents⟩
    ⟨determining one-cusped candidates⟩
    ⟨determining exceptional fillings⟩
```