unhyp

Robert C. Haraway, III*

February 4, 2016

This is a literate `Python` module to determine whether or not a compact orientable 3-manifold[1] with nonempty boundary[2] admits a complete hyperbolic metric on its interior.

The following corollary of Thurston's hyperbolization theorem reduces this determination to a question about the existence of certain surfaces.

Corollary 1. *A compact orientable bounded[3] 3-manifold with $\chi = 0$ is hyp iff it has no faults.*

The words "hyp" and "fault" mean the following.

Definition 2. A compact 3-manifold is *hyp* just when its interior admits a complete, finite hyperbolic metric.

Definition 3. Let s be a properly embedded codimension-1 submanifold of a connected p.l. manifold M. By abuse of notation, also let s denote the image of s in M. Pick a metric on M compatible with its p.l. structure, and let M' be the (abstract, i.e. not in M) path-metric completion of $M \setminus s$.

When M' is disconnected, we say s *separates* M.

When M' has two connected components N, N', we say *s cuts off (an) N from M*. When M is understood from context, we may say *s cuts off an N*, without reference to M. When N, N' are not homeomorphic, we say s cuts off *one* N.

Definition 4. A properly embedded surface s in an orientable 3-manifold M is a *fault* iff $\chi(s) \geq 0$ and it satisfies one of the following:

- s is nonorientable.
- s is a sphere which does not cut off a 3-ball.
- s is a disc which does not off one 3-ball.
- s is a torus which does not cut off a $T^2 \times I$, and does not cut off a ∂-compressible manifold.
- s is an annulus which does not cut off a 3-ball, and does not cut off one $D^2 \times S^1$.

*Research partially supported by NSF grant DMS-1006553.

[1] We work in the PL category throughout.

[2] We require nonempty boundary because Regina implements no tests yet for small Seifert fiberings.

[3] That is, with nonempty boundary.

A proof of Corollary 1 is sketched in [3]. Also found in [3] is the following algorithm, which we implement in Regina:

```
l := fundamental normal surfaces in T
for surf in l:
  if surf is fault:
    return False
T' := finite truncation of T
if T' has a compressing disc:
  return False
l' := vertex Q-normal surfaces in T'
for annulus in l':
  if M has at least two boundary tori:
    if annulus is non-separating:
      return False
  else:
    if annulus is fault:
      return False
else:
  return True
```

Here is an implementation in Regina.

⟨*algorithm for testing hypness*⟩≡

```
def unhypByNormalSurfaces(mfld):
  dno = mfld.getNumberOfBoundaryComponents()
  assert dno > 0
  if not possiblyHyp(mfld):
    return True
  ⟨let T ideally triangulate mfld⟩
  ⟨let l be fundamental surfaces of T⟩
  ⟨for surf in l⟩
    if isFault(surf):
      return True
  ⟨let TT finitely triangulate mfld⟩
  if TT.hasCompressingDisc():
    return True
  ⟨let ll be vertex Q-normal surfaces of TT⟩
  ⟨for surf in ll⟩
    if TT.getNumberOfBoundaryComponents() == 2:
      if isNonSeparatingAnnulus(surf):
        return True
    else:
      if isAnnulusFault(surf):
        return True
  else:
    return False
```

⟨*possibly hyp*⟩≡

```
def possiblyHyp(mfld):
  m = mfld
  return m.isValid() \
     and m.isOrientable() \
     and m.getEulerCharManifold() == 0 \
     and m.isConnected()
```

⟨*let T ideally triangulate mfld*⟩≡

```
T = regina.NTriangulation(mfld)
T.finiteToIdeal()
T.intelligentSimplify()
```

⟨*let l be fundamental surfaces of T*⟩≡

```
nsl = regina.NNormalSurfaceList.enumerate
std = regina.NS_STANDARD
fnd = regina.NS_FUNDAMENTAL
l = nsl(T,std,fnd)
```

⟨*for surf in l*⟩≡

```
n = l.getNumberOfSurfaces()
for i in range(0,n):
  surf = l.getSurface(i)
```

⟨*let TT finitely triangulate mfld*⟩≡

```
TT = regina.NTriangulation(mfld)
TT.idealToFinite()
TT.intelligentSimplify()
```

⟨*let ll be vertex Q-normal surfaces of TT*⟩≡

```
vtx = regina.NS_VERTEX
qd  = regina.NS_QUAD
ll = nsl(TT,qd,vtx)
```

⟨*for surf in ll*⟩≡

```
nn = ll.getNumberOfSurfaces()
for ii in range(0,nn):
  surf = ll.getSurface(ii)
```

We need to implement the predicates "is non-separating annulus," "is fault," and "is $T^2 \times I$."

First, the test for whether or not a surface is a non-separating annulus.

⟨*is non-separating annulus*⟩≡
```
def separates(surf):
  M = surf.cutAlong()
  return not M.isConnected()

def isAnnulus(surf):
  return surf.hasRealBoundary()             \
    and surf.getEulerCharacteristic() == 0 \
    and surf.isOrientable()

def isNonSeparatingAnnulus(surf):
  return isAnnulus(surf) \
     and not separates(surf)
```

Next, the test for whether a surface is an annulus fault.

⟨*is annulus fault*⟩≡
```
def isAnnulusFault(surf):
  return isAnnulus(surf) and \
         isFault(surf)
```

Later it will prove useful to find a non-separating annulus if one exists. So we do that here.

⟨*find non-separating annulus*⟩≡
```
def findNonSeparatingAnnulus(mfld):
  T = mfld
  ⟨let l be fundamental surfaces of T⟩
  a = None
  ⟨for surf in l⟩
    if isNonSeparatingAnnulus(surf):
      a = surf
      break
  return a
```

Now let us implement a fault test. This of course uses tests for 3-ball, ∂-compression, $D^2 \times S^1$, and $T^2 \times I$, so abbreviate these.

⟨*tests*⟩≡
```
b = lambda m: m.isBall()
cd = lambda m: m.hasCompressingDisc()
d2s1 = lambda m: m.isSolidTorus()
t2i = isT2xI
```

There aren't many quick sanity checks to do, so we inline them (so to speak):

⟨*is fault?*⟩≡
```
def isFault(surf):
  s = surf
  x = s.getEulerCharacteristic()
  definitelyNotFault =     \
    x < 0 or               \
    not s.isConnected() or \
    s.isVertexLink()
  if definitelyNotFault:
    return False
```

At this point we know `s` has nonnegative Euler characteristic. If it's not orientable, then it's a fault.

⟨*is fault?*⟩+≡
```
  if not s.isOrientable():
    return True
```

At this point, we know `s` is orientable with nonnegative Euler characteristic. So now we cut along it and see what we get.

⟨*is fault?*⟩+≡
```
  M1 = s.cutAlong()
  M1.intelligentSimplify()
```

If `s` doesn't cut off anything—i.e. if `s` doesn't separate—then it is a fault.

⟨*is fault?*⟩+≡
```
  if M1.isConnected():
    return True
```

Otherwise, it separates, and we should look at the two pieces.

⟨*is fault?*⟩+≡
```
  assert M1.splitIntoComponents() == 2
  M2 = M1.getFirstTreeChild()
  M3 = M2.getNextTreeSibling()
  M2.idealToFinite()
  M2.intelligentSimplify()
  M3.idealToFinite()
  M3.intelligentSimplify()
```

We run the tests now depending on whether `s` is closed or bounded, and depending on what its Euler characteristic is.

When s is a disc,[4] s is a fault when s does not cut off *one* 3-ball. So s is a fault when either both m_2, m_3 are balls or neither m_2 nor m_3 is a ball. That is, when their ballnesses are equal, i.e. $\text{ball}(m_2) = \text{ball}(m_3)$.

⟨*is fault?*⟩+≡
```
    if s.hasRealBoundary():
      if x == 1:
        # s is a disc
        return b(M2) == b(M3)
```

When s is a separating annulus, s is a fault when neither s cuts off *a* 3-ball, nor s cuts off one solid torus.

⟨*is fault?*⟩+≡
```
      else:
        # s had better be an annulus
        assert x == 0
        return not (b(M2) or b(M3)) \
           and d2s1(M2) == d2s1(M3)
```

The rest should be clear.

⟨*is fault?*⟩+≡
```
    else:
      # s is closed
      if x == 2:
        # s is a sphere
        return not (b(M2) or b(M3))
      else:
        # s had better be a torus
        assert x == 0
        return not (t2i(M2) or t2i(M3) \
               or   cd(M2)  or cd(M3))
```

[4]The test for hypness below doesn't ever run this code for discs, since we use `hasCompressingDisc` to find disc faults. But we include this for completeness.

Now for the next part, $T^2 \times I$ detection using Dehn filling.

Definition 5. Suppose M is finitely triangulated. Let T, T' be boundary triangles adjacent along an edge e. Orient e so that T lies to its left and T' to its right.

Let Δ be a fresh tetrahedron, and let τ, τ' be boundary triangles of Δ adjacent along an edge η. Orient η so that τ lies to its left and τ' to its right. Without changing M's topology we may glue Δ to T by gluing η to e, τ to T' and τ' to T. This is called a *two-two* move. The edge η' opposite η in Δ is now a boundary edge of the new finite triangulation.

We say e is *embedded* iff its vertices are distinct. We say e is *co-embedded* or *foldable* iff η' is embedded.

Remark 6. We call a co-embedded edge "foldable" for the following reason. Given a boundary edge e between two boundary triangles T and T', one may glue T to T' and e to itself via a valid, orientation-reversing map, folding them together along e. This gluing will change the topology of M when the vertices opposite e in T and T' are the same vertex. Conversely, when these vertices are distinct, the folding preserves the topology. But the vertices are distinct iff e is co-embedded. Hence the name "foldable."

Notice that folding along a foldable edge decreases the number of boundary triangles, and performing a two-two move on an embedded edge produces a foldable edge and preserves the number of boundary triangles. Therefore, the following `while`-loops terminate:

```
while there's an embedded boundary edge e:
  do a two-two move on e
  while there's a foldable boundary edge f:
    fold along f
```

The obvious postcondition of the `while` loop is that there's no embedded boundary edge. Since the boundary is still triangulated, this is equivalent to each boundary component having only one vertex on it. Since each boundary component is a torus, $V - E + F = 0$. Now, $V = 1$, and since the cellulation is a triangulation, $3 * F = 2 * E$.

$$\begin{aligned} 1 - E + F &= 0 \\ 2 - 2 * E + 2 * F &= 0 \\ 2 - 3 * F + 2 * F &= 0 \\ 2 - F &= 0 \\ 2 &= F, \end{aligned}$$

and there are only two triangles. We may fill any cusp we like by folding along one of the remaining three (non-foldable) edges.

Remark 7. The routine in `SnapPea` is more complicated because, rather than filling in a cusp any old way, `SnapPea` wants to make sure the filling compresses some given slope in the cusp.

To implement this algorithm, let us take stock of the tools `Regina` provides.

The algorithm is centered around edges, and instances of `Regina`'s class `NEdge` represent edges. `NEdge` has the following methods:

- `isBoundary`
- `getVertex`
- `getTriangulation`
- `getEmbeddings`

Except for the first, they all do more or less what you would expect. More specifically, if `e` instantiates `NEdge` and represents an edge e in a triangulation T, then `e.isBoundary()` is `True` iff e is a boundary edge; `e.getVertex(0)` instantiates `NVertex` and represents the source of e, whereas `e.getVertex(1)` represents its sink; and `e.getTriangulation()` is instantiates `NTriangulation` and represents T.

In particular, here's a method to determine whether or not an edge is embedded.

⟨*embedded edge?*⟩≡
```
def embedded(edge):
  src = edge.getVertex(0)
  snk = edge.getVertex(1)
  return src != snk
```

`getEmbeddings` is more complicated, but also more important. `e.getEmbeddings()` is a list (in order) of instances of the class `NEdgeEmbedding`. Let `phi` be an element of `e.getEmbeddings()`. `phi` represents an embedding ϕ of e into an incident tetrahedron Δ.

Some instance `phi.getTetrahedron()` of `NTetrahedron` represents Δ. Each instance `D` of `NTetrahedron` comes with a method `D.getVertex`, which represents an identification $j_\Delta : \mathbf{4} \to \Delta^0$ of $\mathbf{4} = \{0, 1, 2, 3\}$ with the set Δ^0 of vertices of Δ.

`phi.getVertices()`, perhaps confusingly, is an instance of `NPerm4`. General instances of `NPerm4` represent elements of $S_\mathbf{4}$. `phi.getVertices()` actually represents an element $f \in A_\mathbf{4}$, the alternating group on $\mathbf{4}$, with the following property. Let s, s' be the source and sink of e. Then $\phi(s) = j_\Delta(f(0))$ and $\phi(s') = j_\Delta(f(1))$.

We note that there are two elements of $S_\mathbf{4}$ satisfying these equations, but only one in $A_\mathbf{4}$. We choose the one in $A_\mathbf{4}$.

`e.getEmbeddings()` is ordered so that we can give a consistent orientation to the edges opposite e, in the following way:

> Let `l = e.getEmbeddings()` have length n; for all i with $0 \leq i < n$, let `l[i]` represent the embedding ϕ_i; let `l[i].getTetrahedron()` represent Δ_i; let $j_i = j_{\Delta_i}$; and let `l[i].getVertices()` represent f_i. Then for all i with $1 \leq i < n - 1$, the vertices $j_{i-1}(f_{i-1}(3))$ and $j_i(f_i(2))$ are glued in the triangulation.

Furthermore, if e is a boundary edge, then each of `l[0]` and `l[-1]` (i.e. the last element of `l`) represents an embedding of e into the boundary. In particular, if $j_0(f_0(2))$ becomes the vertex v in the triangulation and $j_{-1}(f_{-1}(3))$ becomes w, then v and e determine a boundary face, and so do w and e.

We should finally explain another method `NTetrahedron` provides, namely `joinTo`. Suppose Δ, H are tetrahedra. How shall we join them up? Glue them along faces. How shall we name faces? Call faces by their opposite vertices. How shall we determine a gluing map? Well, let p be a gluing map from the face v_* opposite v in Δ to the face w_* opposite w in H. Then p restricted to the vertices of the faces has a unique extension to a bijection $P : \Delta^0 \to H^0$. Then we get a uniquely determined element $\sigma_p = j_H^{-1} \circ P \circ j_\Delta$ of $S_\mathbf{4}$.

Conversely, for any such element $s \in S_\mathbf{4}$, there is a unique affine map $\pi_s : v_* \to w_*$ such that $\sigma_{\pi_s} = s$.

So we may determine a gluing by

- which tetrahedra we're gluing,
- which faces are getting glued, and
- what is the associated element of $S_\mathbf{4}$.

In `Regina` it's more spartan than that. First of all, every instance of `NTetrahedron` has the child method `joinTo`. There's no need to include that instance as an argument to the procedure; `joinTo` implicitly regards its parent tetrahedron as Δ above. Also, given the permutation and the face on Δ to glue, the face on H is determined, so that face need not be included as an argument to `joinTo`.

In conclusion,

```
D.joinTo(i,E,s)
```

is the `Regina` syntax for gluing tetrahedron `D` to a tetrahedron `E` by gluing the face in `D` opposite `D.getVertex(i)` to the face in `E` opposite `E.getVertex(s(i))` by the map determined by `s`.

Therefore, we care primarily about tetrahedra and permutation representatives, and not so much about the edge embeddings ϕ_i themselves. In fact, since ultimately we're only concerned with boundary edges, all we care about are f_0, f_{-1}, Δ_0, and Δ_{-1}. So let's write methods to return their representatives.

⟨*left and right maps and tets*⟩≡
```
def lrmaps(edge):
  embs = edge.getEmbeddings()
  return (embs[0].getVertices(),\
          embs[-1].getVertices())

def lrtets(edge):
  embs = edge.getEmbeddings()
  return (embs[0].getTetrahedron(),\
          embs[-1].getTetrahedron())
```

We've already written a method for determining whether or not an edge is embedded. Let's write a method to determine whether or not it's coembedded.

The definition we gave already was quick, but implementing it is altogether unnecessary, for with the terminology we have now, there is a better characterization of foldability. First of all, an edge had better be a boundary edge if it is going to be foldable.

⟨*foldability*⟩≡
```
def foldable(edge):
  if not edge.isBoundary():
    return False
```

As we've set it up, the tetrahedra and permutations are as follows:

⟨*foldability*⟩+≡
```
  (F0,F_1) = lrmaps(edge)
  (D0,D_1) = lrtets(edge)
```

Now, an edge e is coembedded iff it becomes embedded after a two-two move. This is equivalent to saying that the vertices opposite e in the boundary faces are distinct. But these vertices are $j_0(f_0(2))$ and $j_{-1}(f_{-1}(3))$. So we can just test equality of these.

⟨*foldability*⟩+≡
```
  lvx = D0.getVertex(F0[2])
  rvx = D_1.getVertex(F_1[3])
  return lvx != rvx
```

Having finished the foldability test, let us now implement the next part of the algorithm, viz. a two-two move on a boundary edge. This attaches a fresh tetrahedron t to the triangulation of the edge.

⟨*two-two move*⟩≡
```
def twoTwo(edge):
  M = edge.getTriangulation()
  T = M.newTetrahedron()
```

Use the same notation as above.

⟨*two-two move*⟩+≡
```
  (F0,F_1) = lrmaps(edge)
  (D0,D_1) = lrtets(edge)
```

Now the faces adjacent to e are opposite the vertices $j_0(f_0(3))$ and $j_{-1}(f_{-1}(2))$. We wish to glue to these faces the two faces of t that include both $j_t(0)$ and $j_t(1)$. These are the faces opposite $j_t(2)$ and $j_t(3)$.

To determine what the gluing maps should be, we may just follow the lead of `getEdgeEmbedding` and insist that $j_0(f_0(3))$ get glued to $j_t(2)$ and $j_{-1}(f_{-1}(3))$ to $j_t(3)$. (If such insistence isn't convincing, draw a picture.)

The first gluing map also sends $j_t(0)$ to $f_0(0)$ and $j_t(1)$ to $f_0(1)$. So the associated permutation s_0 is plainly f_0 precomposed with the cycle $x = (2\ 3)$. The same is true for the other gluing map's permutation s_{-1}, except with f_{-1} instead of f_0.

⟨*two-two move*⟩+≡
```
  X = regina.NPerm(2,3)
  S0 = F0 * X
  S_1 = F_1 * X
  T.joinTo(2,D0,S0)
  T.joinTo(3,D_1,S_1)
```

The last nontrivial operation we must implement is to fold along a boundary edge. We must first insist that the edge be a boundary edge.

⟨*fold along a boundary edge*⟩≡
```
def foldAlong(edge):
  assert edge.isBoundary()
```

Use the same notation as before.

⟨*fold along a boundary edge*⟩+≡
```
  (D0,D_1) = lrtets(edge)
  (F0,F_1) = lrmaps(edge)
```

The face ℓ of Δ_0 to be glued is the face opposite $j_0(f_0(3))$, and the face ℓ' of Δ_{-1} to be glued is the face opposite $j_{-1}(f_{-1}(2))$. So to glue ℓ to ℓ', we need a permutation fixing 0 and 1, and taking $f_0(3)$ to $f_{-1}(2)$. This permutation is $f_{-1} \circ (2\ 3) \circ f_0^{-1}$.

⟨*fold along a boundary edge*⟩+≡
```
  X = regina.NPerm(2,3)
  glu = F_1 * X * F0.inverse()
  D0.joinTo(F0[3], D_1, glu)
```

This concludes the difficult portion of the implementation. The rest is simple.

The first and second `while` loops don't have implementations as such in `Python`. We can simulate them by implementing an operation that returns either the first boundary edge satisfying a predicate, or the `Python` primitive `None` if there is no such edge.

⟨*first boundary edge*⟩≡
```
def firstBoundaryEdge(mfld,pred):
  cpts = mfld.getBoundaryComponents()
  for d in cpts:
    n = d.getNumberOfEdges()
    for i in range(0,n):
      e = d.getEdge(i)
      if pred(e):
        return e
  else:
    return None
```

This uses the `Python` idiom of an `else` clause after a `for` loop.

Now we are ready to implement the `while` loops.

⟨*simplify cusps*⟩≡
```
def simplifyCusps(finite_mfld):
  M = finite_mfld
  fBE = firstBoundaryEdge
  f = fBE(M,embedded)
  while f != None:
    twoTwo(f)
    g = fBE(M,foldable)
    while g != None:
        foldAlong(g)
        g = fBE(M,foldable)
    f = fBE(M,embedded)
```

To implement the $T^2 \times I$ test, now we just need to implement a routine as described in [3]. We need to first make sure the manifold is irreducible and ∂-incompressible. There isn't at present an explicit method for irreducibility of bounded manifolds in Regina, so we roll our own very slow method. We define `isFault` below, which is allowed in `Python`.

$\langle$*is* $T^2 \times I$*?*$\rangle\equiv$

```
def irreducible(regina_mfld):
  M = regina.NTriangulation(regina_mfld)
  M.finiteToIdeal()
  M.intelligentSimplify()
  nsl = regina.NNormalSurfaceList.enumerate
  l = nsl(M, regina.NS_STANDARD,  \
             regina.NS_FUNDAMENTAL)
  n = l.getNumberOfSurfaces()
  for i in range(0,n):
    s = l.getSurface(i)
    x = s.getEulerCharacteristic()
    if x != 2:
      continue
    if isFault(s):
      return False
  else:
    return True

def isT2xI(regina_mfld):
  M = regina.NTriangulation(regina_mfld)
  M.finiteToIdeal()
  M.intelligentSimplify()
```

There are some sanity checks we can run here before the irreducibility and ∂-incompressibility tests to save time—e.g. whether the manifold is connected, whether it has two torus boundary components, and so on. Put these sanity checks under the umbrella function `possiblyT2xI`, and implement this later.

$\langle$*is* $T^2 \times I$*?*$\rangle{+}{\equiv}$

```
    if not possiblyT2xI(M):
      return False
```

Now we implement our test. We point out that since we end up simplifying the cusps and doing three fillings, we make a clone of N of M, simplify N's cusps, then clone N three times before filling along the three slopes. We also note that `simplifyCusps` simplifies all boundary components' triangulations as specified in [3].

$\langle$*is* $T^2 \times I$*?*$\rangle{+}{\equiv}$

```
    if not irreducible(M):
      return False
    if M.hasCompressingDisc():
      return False
    a = findNonSeparatingAnnulus(M)
    if a == None:
      return False
    mu = a.cutAlong()
    mu.intelligentSimplify()
    if not mu.isSolidTorus():
      return False
    D = regina.NTriangulation(M)
    simplifyCusps(D)
    T = D.getBoundaryComponent(1)
    n = T.getNumberOfEdges()
    assert n == 3
    for i in range(0,n):
      clone = regina.NTriangulation(D)
      cpt = clone.getBoundaryComponent(1)
      e = cpt.getEdge(i)
      foldAlong(e)
      if not clone.isSolidTorus():
        return False
    else:
      return True
```

Now we should get around to implementing the remaining sanity checks.

⟨*possibly* $T^2 \times I$⟩≡
```
def possiblyT2xI(mfld):
  m = mfld
  dno = m.getNumberOfBoundaryComponents()
  if not (possiblyHyp(m) and dno == 2):
    return False
  cpts = m.getBoundaryComponents()
  for d in cpts:
    x = d.getEulerCharacteristic()
    if x != 0:
      return False
  H1 = m.getHomologyH1()
  if not H1.toString() == '2 Z':
    return False
  H2 = m.getHomologyH2()
  if not H2.isZ():
    return False
  H1R = m.getHomologyH1Rel()
  if not H1R.isZ():
    return False
  return True
```

⟨*has torus boundary?*⟩≡
```
def hasTorusBoundary(mfld):
  m = mfld
  if m.isClosed():
    return False
  for v in m.getVertices():
    if not v.getLink() == regina.NVertex.TORUS:
      return False
  return True
```

Finally, the code at present has the following flaw: that it verifies hyperbolicity by enumerating all fundamental normal surfaces of a certain flavor, and checking that they are not faults. This takes a long time, and there is now a better routine which may verify hyperbolicity. This method is part of `Regina`; it detects whether a triangulation admits a strict angle structure, existence of which is a sufficient condition for hyperbolicity.

⟨*is hyp?*⟩≡
```
def isHyp(regina_manifold):
  m = regina.NTriangulation(regina_manifold)
  m.intelligentSimplify()
  if m.hasStrictAngleStructure():
    return True
  else:
    return not unhypByNormalSurfaces(m)
```

⟨*unhyp.py*⟩≡
```
import regina
⟨possibly hyp⟩
⟨is non-separating annulus⟩
⟨is annulus fault⟩
⟨find non-separating annulus⟩
⟨embedded edge?⟩
⟨left and right maps and tets⟩
⟨foldability⟩
⟨two-two move⟩
⟨fold along a boundary edge⟩
⟨first boundary edge⟩
⟨simplify cusps⟩
⟨possibly T^2 × I⟩
⟨has torus boundary?⟩
⟨is T^2 × I?⟩
⟨tests⟩
⟨is fault?⟩
⟨algorithm for testing hypness⟩
⟨is hyp?⟩
```

References

[1] F. Bonahon, *Geometric structures on 3-manifolds*, Chapter 3 from *Handbook of Geometric Topology*, ed. R. J. Daverman and R. B. Sher, Elsevier, New York, 2002.

[2] B. A. Burton, R. Budney, W. Pettersson, et al., `Regina`*: software for 3-manifold topology and normal surface theory*, http://regina.sourceforge.net/ , 1999–2014.

[3] Robert C. Haraway, III, *Determining hyperbolicity of multiply-cusped 3-manifolds*, preprint available at author's website.

[4] W. Jaco and J. L. Tollefson, *Algorithms for the complete decomposition of a closed 3-manifold*, Illinois J. of Math., **39**:3, Fall 1995, pp. 358–406.

[5] M. Kapovich, *Hyperbolic manifolds and discrete groups*, Birkhäuser, Boston, 2001.

[6] A. Marden, *Outer circles: an introduction to hyperbolic 3-manifolds*, Cambridge University Press, New York, 2007.

[7] S. Matveev, *Algorithmic topology and classification of 3-manifolds*, Springer-Verlag, New York, 2003.

[8] W. P. Thurston, *Three-dimensional manifolds, Kleinian groups and hyperbolic geometry*, Bull. Amer. Math. Soc. (N.S.) **6**:3 (1982), 357–381.

[9] J. L. Tollefson, *Normal surface Q-theory*, Pacific J. of Math., **183**:2 (1998), 359–374.