Billiard Dynamics on Surfaces of Revolution

Isaac Garfinkle

September 5, 2016

1 Billiard Tables

A *billiard table* is a closed region on a surface with piecewise C^2 boundary. Inside the region, a particle may move in a straight line with constant unit speed (akin to a ball in the game of billiards.) Upon reaching the boundary of the table, the particle reflects without losing speed, where the angle of incidence equals the angle of reflection. This gives rise to a continuous dynamical system, where, given a point in the region an an initial direction of the particle's velocity, we can find where the particle will be and its direction after time (or distance) t. This is called the Billiard Flow of the table. Each line between collisions is called a trajectory, and the movement of a particle over time is called its orbit.

With this definition, there are some questions that must be resolved. Since the boundary of the table is only piecewise C^2, rather than completely C^2, it may be that our boundary has corners. Since reflections at corners are often ill-defined, we will simply ignore any orbit that would ever reflect at a corner. Luckily, this represents only small (measure zero) set of all orbits, so we don't lose a whole lot of information. Similarly, we will ignore any orbit that eventually intersects the boundary with angle zero (such as if a trajectory was along a tangent line to a circle), since it is unclear whether this reflection should be counted.

Having dealt with all the edge cases, we can define the *billiard map.* The phase space of a billiard table is a two dimensional table representing all the trajectories on the table, assuming the table boundary is oriented. The first coordinate gives the starting point of a trajectory on the boundary of the table, and the second coordinate gives the angle from the positive direction of the boundary that the trajectory starts. This will always be between 0 and π, where, for example, a value of $\pi/2$ means the trajectory points perpendicular to the boundary at that point. The Billiard Map sends each trajectory to the next trajectory in that orbit. From a given starting trajectory, we allow the particle to flow until it reaches another boundary. The particle reflects, and this gives us another trajectory, the result of the billiard map. So each orbit has a doubly infinite sequence of trajectories ascribed to it (since we can just as easily flow backward), each of which is determined by a location on the boundary and an angle of reflection.

If we put a particle on a square billiard and trace out its orbit over time, a pattern will emerge. The slope of all the trajectories is the same (up to sign). This pattern can be formalized with the notion of integrability. A billiard table B is said to be integrable if there is a piecewise continuous, non-constant function $F : \partial B \times [0, \pi] \to \mathbb{R}$ on the phase space whose level sets are one-dimensional T-invariants that foliate (fill) the phase space.

For example, on the square billiard, we can choose this function F to map each point in the phase space to the absolute value of its slope. This quantity is preserved by the billiard map, it is continuous except at the four corners, and its level sets are line segments in the phase space, so this is an integrating function. Similarly, on the circle billiard, the angle of reflection will not change, so $F(b, \theta) = \theta$ is a suitable choice for integrating function.

2 Surfaces of Revolution

While billiards are traditionally thought of as being in the plane, this does not need to be the case. Any smooth surface can be used to form billiard tables. Billiards on surfaces with constant Gaussian curvature, specifically curvature 1, -1, and 0 (the sphere, the hyperplane, and the plane, respectively) are well studied. However, there is far less work on billiards on surfaces with nonconstant curvature. Our work focuses on surfaces of revolution, which are an extension of surfaces with constant curvature (there are surfaces of revolution with any given fixed Gaussian curvature.)

Formally, a surface of revolution is defined by a parametric plane curve $(f(t), g(t))$ where $f(t)$ is positive (so the curve lives in the right half-plane.) From this we get a surface of revolution parameterized by

$$r(u, v) = (f(v)\cos(u), f(v)\sin(u), g(v)).$$

Notable curves on surfaces of revolution are parallels, which are curves with a fixed v and are thus shaped like circles, and meridians, which are curves with a fixed u, and are thus shaped like the plane curve (f, g). Since we will be dealing with particles moving on such surfaces, it is important to note what a geodesic (intuitively, a straight line) will look like on such a surface.

On an arbitrary surface embedded in $\mathbb{R}^3$, a geodesic is a curve whose second derivative is perpendicular to the surface. On surfaces of revolution, geodesics have the property that the Clairaut function $r\cos(\theta)$ is constant along them, where r is the radius $f(v)$ of a point, and θ is the angle between the tangent line to the geodesic and the tangent line to the parallel running through that point. It is important to note that curves having this property may not be geodesics. For example, the parallel at height v_0 will have constant Clairaut function $f(v_0)$, but it will only be a geodesic if $f'(v_0) = 0$. On the other hand, all meridians are geodesics (with Clairaut constant 0).

3 Integrability on Surfaces of Revolution

One of the challenges in studying billiards on surfaces of revolution is that many types of tables that are integrable on surfaces like the plane are very difficult to define the boundary of on a surface of revolution. The primary example of this is a circle. A circle, defined as the set of points a fixed distance from a given point, is easy to parameterize on the plane or the sphere, but is very challenging to parameterize on a surface of revolution. While circles on the plane or the sphere will have constant geodesic curvature, on an arbitrary surface, this is not the case. For this reason, we turn our attention instead to polygons, tables whose boundary is made up of geodesic components.

One table that is integrable on a surface of revolution is a rectangle, bounded by parallels on the top and bottom, and meridians on the sides. The absolute value of the Clairaut constant serves as an integrating function, since reflections on one of the parallels change the angle θ to $-\theta$ and reflections on the meridians change the angle θ to $\pi - \theta$. Since the Clairaut function is preserved along each trajectory, and on this table is preserved along reflections, it will be preserved along the whole billiard map. Unfortunately, this strategy of using the Clairaut constant to find integrating functions does not get us much farther than for particular examples like these rectangles. If a trajectory intersects an arbitrary geodesic boundary, there is no equation relating the three Clairaut constants that does not also depend on the radius of the surface at the point of intersection. If θ_B is the angle of the boundary from a parallel, θ_- is the angle of the incoming trajectory, and θ_+ is the angle of the outgoing trajectory, we have the relation $\theta_+ = 2\theta_b - \theta_-$, and we see there is a nice relation between θ_+ and θ_- if θ_b is 0 (as in a parallel) or $\pi/2$ (as in a meridian.) Letting C_+, C_-, and C_b be the respective Clairaut constants, we see $C_+ = r\cos(2\theta_b - \theta_-)$, and so far we have been unable to reduce this equation to depend only on C_+, C_-, and C_b and not on the radius r.

Our research focused on what we found to be the next simplest thing. We explored biangles, or 2-gons, on surfaces of revolution. Specifically, we chose one boundary component (henceforth, the *base*) to be a parallel geodesic, that is, a parallel at height v_0 where $f'(v_0) = 0$. The other boundary component (the *top*) a geodesic that intersects the base twice, so the resulting table looks like a wedge. For the top component to "turn around" and reintersect the base, there must be a point on the surface with radius equal the the Clairaut constant of the top component. This ensures that the top geodesic eventually levels off and turns around back toward the base. Some biangles that we tested appeared to be integrable, while others did not. We seek criteria for such a billiard table to be integrable.

4 Examples

In this section we present a number of examples of billiard tables on surfaces of revolution. Each example will focus on one billiard table, and will present traces of various orbits as well as diagrams of orbits on the phase space.

4.1 Rectangle

Surface given by $f(v) = 1 - v^2$, $g(v) = v$. Table bounded by $0 < u < \pi/2$ and $0 < v < 1/2$.

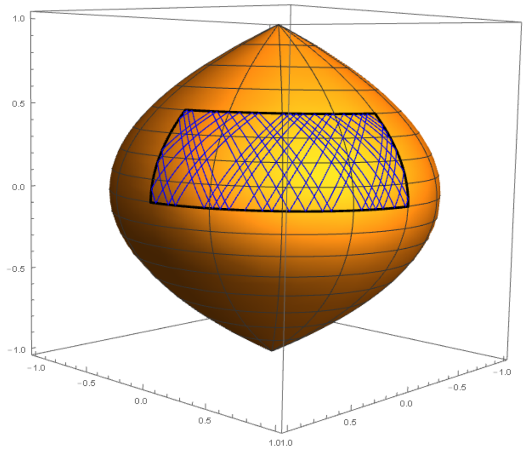

Figure 1: Initial trajectory in the middle of bottom component with initial angle $\pi/3$.

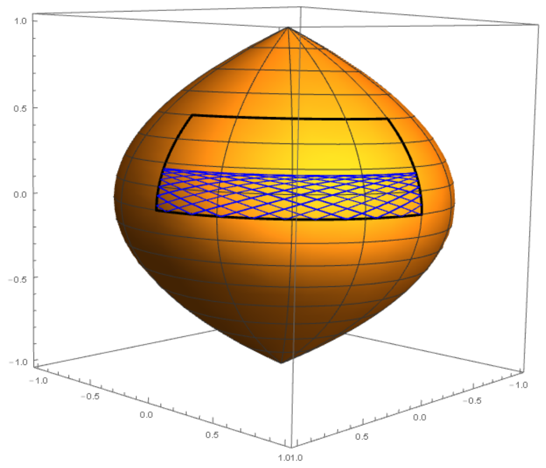

Figure 2: Initial trajectory in the middle of bottom component with initial angle $\pi/10$.

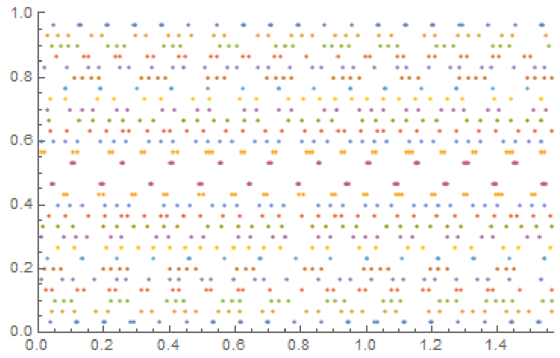

Figure 3: Phase space of bottom component

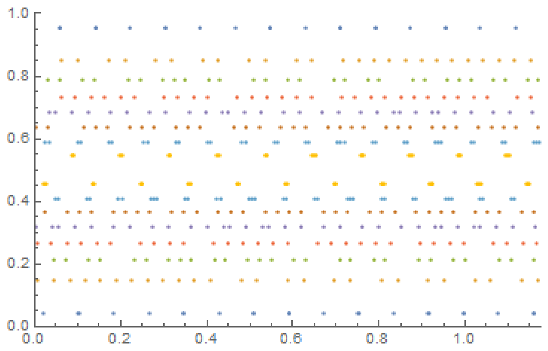

Figure 4: Top component

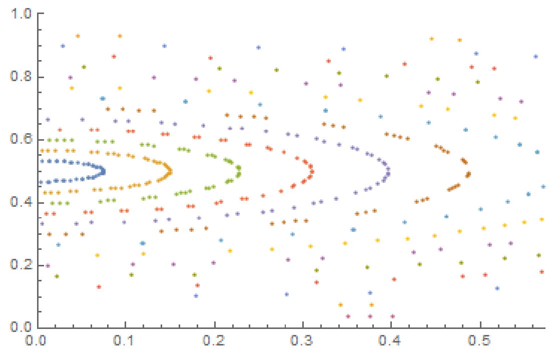

Figure 5: Left component

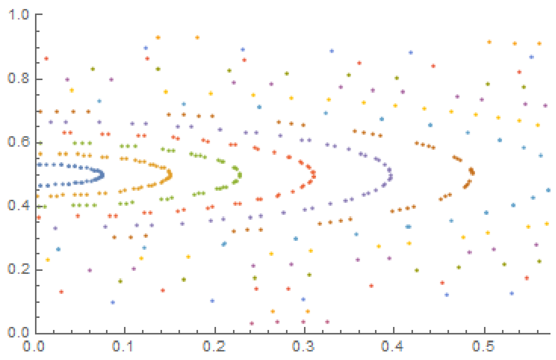

Figure 6: Right component

4.2 Biangle 1

Surface given by $f(v) = 1 - v^2$, $g(v) = v$. Table bounded by $v = 0$ and a geodesic meeting the base at angle $\pi/4$

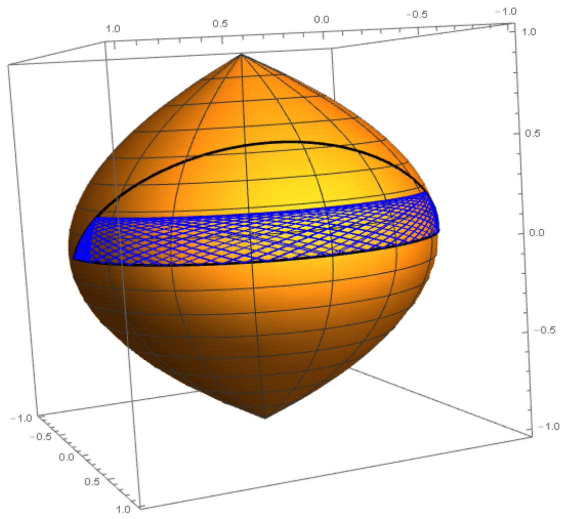

Figure 7: Initial trajectory in the middle of bottom component with initial angle $\pi/10$.

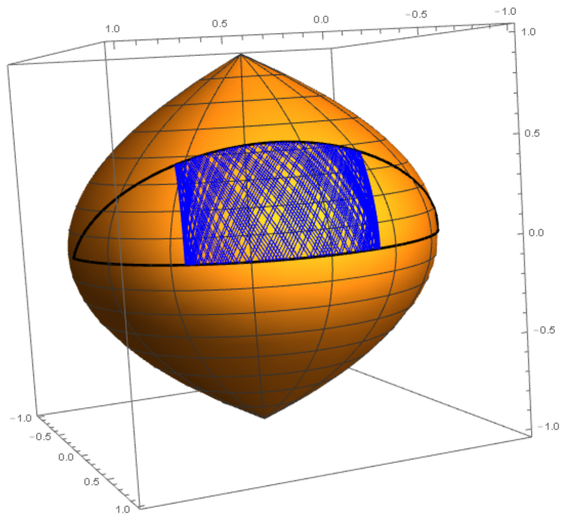

Figure 8: Initial trajectory in the middle of bottom component with initial angle $\pi/3$.

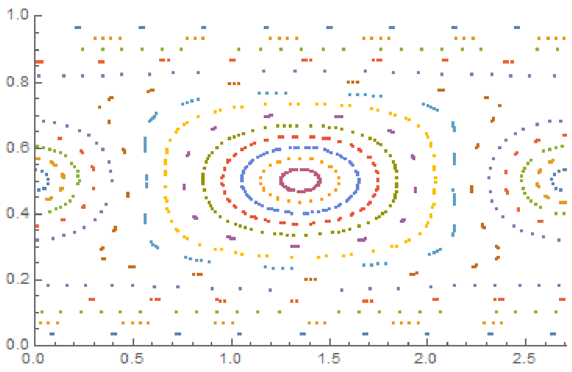

Figure 9: Phase space of bottom component

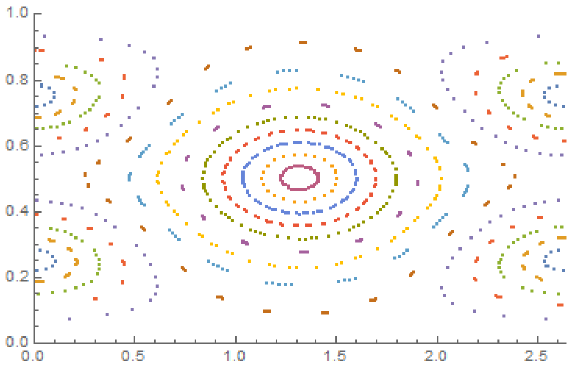

Figure 10: Top component

4.3 Biangle 2

Surface given by $f(v) = (1 - v^2)^5$, $g(v) = v$. Table bounded by $v = 0$ and a geodesic meeting the base at angle $2\pi/5$

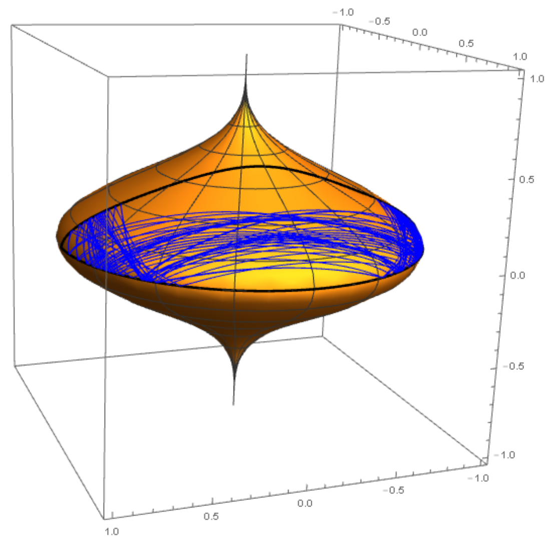

Figure 11: Initial trajectory $\pi/2$ along bottom component with initial angle $pi\pi/10$.

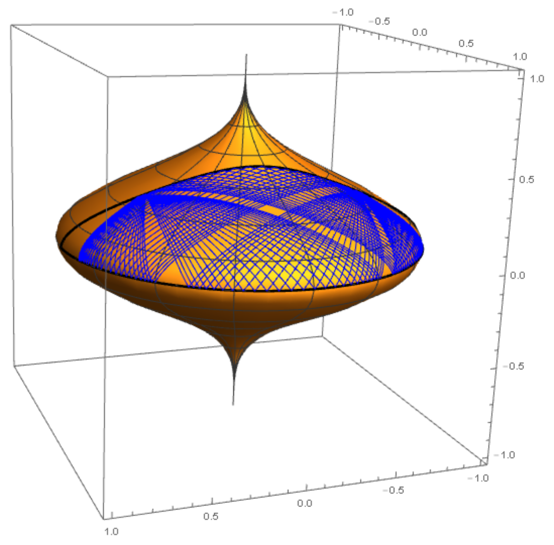

Figure 12: Initial trajectory $\pi/2$ along bottom component with initial angle $pi\pi/3$.

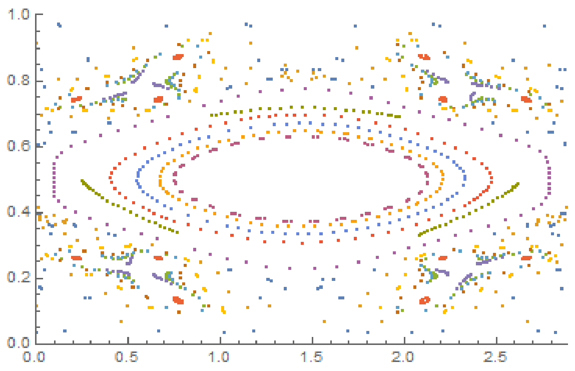

Figure 13: Phase space of bottom component

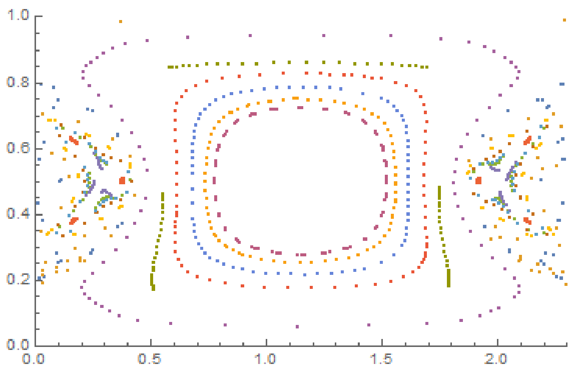

Figure 14: Top component

As can be seen in the diagrams of the phase space, while some regions appear to be integrable (like the centers of the plots), it is less clear that the whole table is integrable. While the orbit in Figure 12 looks regular and patterned, there is less visible order in the orbit in Figure 11.

4.4 Biangle 3

Surface given by $f(v) = (1 - v^2)^{0.65}$, $g(v) = v$. Table bounded by $v = 0$ and a geodesic meeting the base at angle $\pi/3$

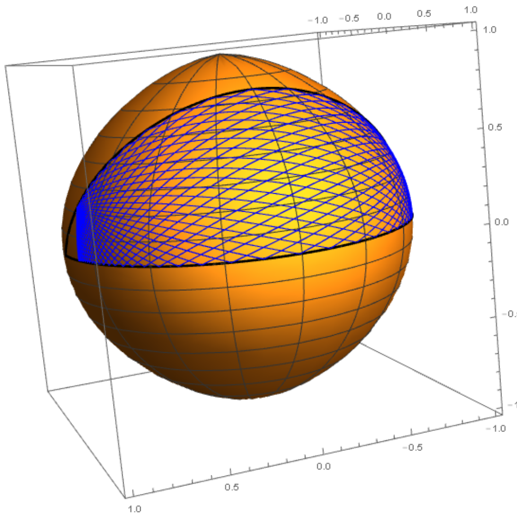
Figure 15: Initial trajectory in the middle of bottom component with initial angle $\pi/10$.

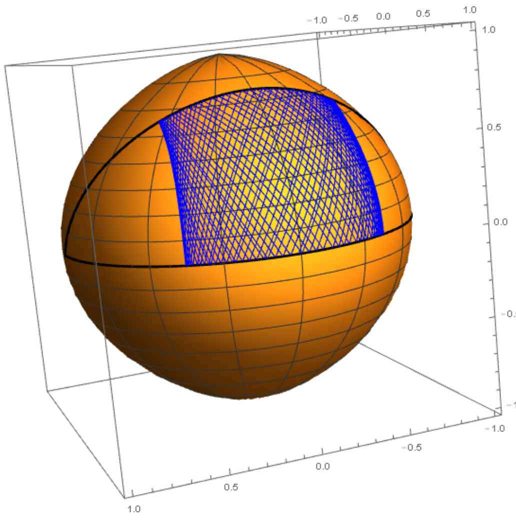
Figure 16: Initial trajectory in the middle of bottom component with initial angle $\pi/3$.

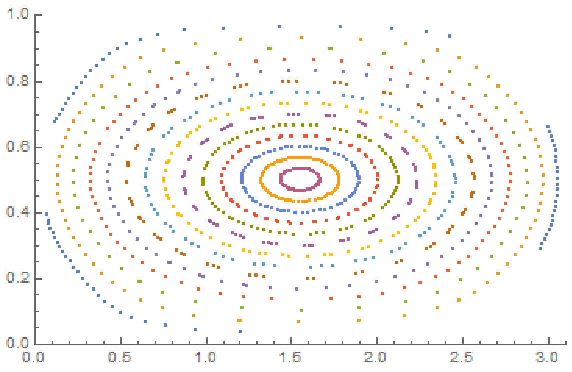
Figure 17: Phase space of bottom component

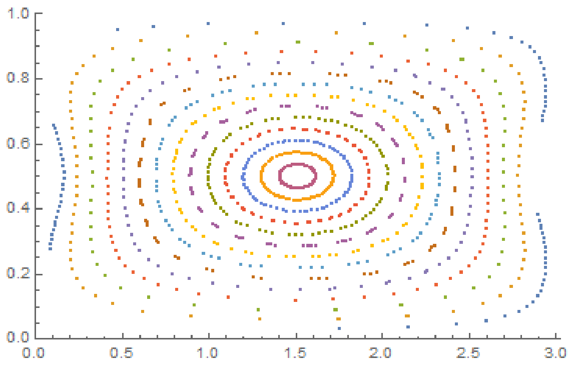
Figure 18: Top component

5 Questions and Future Work

One phenomenon we found was that obviously non-integrable biangles only appeared on surfaces with negative curvature. This leads us to ask whether every biangle table on a surface of revolution with strictly positive curvature in its interior is integrable. Furthermore, do biangles with some negative curvature have regions of integrability in their phase space?

Except the sphere, all examples we observed had the potential for a trajectory to start and end on the same boundary component. On the sphere, this can never be the case, since a biangle can be rotated to be a wedge bounded by two meridians, and every trajectory must go from one side to the other. We believe surfaces of constant positive curvature are the only surfaces with this property.

Conjecture 1. *If a biangle on a surface of revolution has the property that every trajectory from one side must intersect the other side, then the surface must have constant curvature in the interior of the biangle.*

If a biangle has this property, it must be that all trajectories from one corner intersect the other corner. Otherwise, there would be a trajectory from a corner to one of the sides, and thus there would have to be a nearby trajectory that starts on that side and ends on that same side. Furthermore, the restriction to surfaces of revolution avoids the case where a biangle has mostly constant Gaussian curvature on its interior, save some small region with a wildly different curvature. In surfaces of revolution, Gaussian curvature is constant along parallels, so we won't have small regions of different curvature. It would appear from the examples included above and from others we have run that if the Gaussian curvature is larger near the base of the biangle, there will be trajectories that start and end at the base, whereas if the curvature is smaller near the base, there will be trajectories that start and end on the top.

Additionally, we have explored the relationship between integrability and hyperbolicity on surfaces of revolution. Informally, hyperbolicity describes the sensitivity of the billiard map subject to small changes in initial conditions. Formally, the hyperbolicity of an orbit is the trace of the derivative of the geodesic flow along the orbit. An orbit is said to be hyperbolic if the absolute value of this trace exceeds two. One might assume that apparently integrable billiards would not be hyperbolic, but this is not the case. Many biangles that appear integrable also appear to have hyperbolic orbits.

6 Appendix: Mathematica code

```
    (*Gives the geodesic with starting coordinates u0,v0 at t=0 and angle \
\[Theta].*)
geodesicSolve[u0_, v0_, \[Theta]_] :=
  NDSolve[{u''[t] + (2*f[v[t]] (D[f[v], v] /. v -> v[t]))/f[v[t]]^2 *
        u'[t] v'[t] == 0,
     v''[t] -
       (f[
           v[t]] (D[f[v], v] /.
             v -> v[t]))/((D[f[v], v] /.
              v -> v[t])^2 + (D[g[v], v] /. v -> v[t])^2) (u'[t])^2 +
        ((D[f[v], v] /. v -> v[t])*(D[f[v], v, v] /.
              v -> v[t]) + (D[g[v], v] /.
              v -> v[t])*(D[g[v], v, v] /.
              v -> v[t]))/((D[f[v], v] /.
              v -> v[t])^2 + (D[g[v], v] /. v -> v[t])^2) (v'[t])^2 ==
        0,
      u[0] == u0,
      u'[0] == Cos[\[Theta]]/f[v[0]],
      v[0] == v0,
      v'[0] == Sin[\[Theta]]/Sqrt[f'[v[0]]^2 + g'[v[0]]^2]},
     {u, v},  {t, -100, 100}][[1]];

(*Gives the equation for a parallel on the SoR with given height and \
starting angle. The curve will have unit speed*)
getParallel[height_,
  startingAngle_] := {u -> Function[t, t/(f[height]) + startingAngle],
   v -> Function[t, height]}

(*Due to slight errors in machine precision, some numbers slightly \
outside [-1,1] may need to be adjusted to within those bounds so that \
inverse trig functions with those values do not produce complex \
outputs (see below).*)
sanitize[x_] := Piecewise[{{1, x > 1}, {-1, x < -1}}, x]

(*Gives angle mod 2Pi from Sine and Cosine*)
getAngleFromCartesianCoords[x_, y_] :=
 Piecewise[{{ArcCos[sanitize[x/Sqrt[x^2 + y^2]]],
    ArcSin[sanitize[y/Sqrt[x^2 + y^2]]] >= 0}}, -ArcCos[
    sanitize[x/Sqrt[x^2 + y^2]]]]

(*If you need to specify where the program should start looking for \
such an intersect (since it is found by approximation), you can do \
that too*)
(*Known problem: We should want that the u values are equal mod 2Pi, \
but adding this condition causes problems somehow*)
getIntersect[c1_, c2_, guess1_, guess2_] :=
 FindRoot[{(u[t] /. c1) == (u[t2] /. c2), (v[t] /. c1) == (v[t2] /.
      c2)}, {{t, guess1}, {t2, guess2}}]

(*Angle of curve to parallel at given point*)
getAngleFromCurve[curve_, t_] :=
 getAngleFromCartesianCoords[(u'[t] /. curve)*
   f[v[t] /. curve], (v'[t] /. curve)*
   Sqrt[(f'[v[t] /. curve])^2 + (g'[v[t] /. curve])^2]]

(*Given a trajectory, a boundary, and the intersect (previously \
calculated) where the two curves meet, this gives the next trajectory \
after reflection*)
getNextTrajectory[curve_, boundary_, intersect_] :=
 Module[ {t1, tb, fv, sp, v0, u0}, t1 = t /. intersect[[1]];
  tb = t2 /. intersect[[2]];
  v0 = v[t1] /. curve;
  fv = f[v0];
  sp = Sqrt[(f'[v0])^2 + (g'[v0])^2];
  Return[geodesicSolve[u[t1] /. curve,
    v[t1] /. curve, -getAngleFromCurve[curve, t1] +
     2 getAngleFromCurve[boundary, tb]]]]

(*Returns plot of curve with given length, thickness, and color*)
getPlot[curve_, length_, thickness_, color_] :=
 ParametricPlot3D[{Evaluate[{f[v[t]] Cos[u[t]], f[v[t]] Sin[u[t]],
      g[v[t]]} /. curve]} , {t, 0, length},
  PlotStyle -> {Thickness[thickness], color}]

(*These color and thicken automatically based on type of curve*)
getBoundaryPlot[curve_, length_] :=
 getPlot[curve, length, 0.005, Black]
getTrajectoryPlot[curve_, length_] :=
 getPlot[curve, length, 0.002, Blue]


(*Calculates information about the next reflection of the curve with \
the table defined by the boundary components listed*)
nextIntersectDetailed[curve_, boundaryList_] :=
 Module[{bestBoundary, thisT, thisTBoundary, bestIndex, bestT, int, j,
    k, l, bestInt},
  bestBoundary = boundaryList[[1]][[1]];
  thisT = 0;
  bestIndex = 0;
  bestT = 9999;
  For[j = 1, j <= Length[boundaryList], j++,
   For[k = Floor[boundaryList[[j]][[2]]],
     k < Ceiling[boundaryList[[j]][[3]]],
     k = k + (boundaryList[[j]][[3]] - boundaryList[[j]][[2]])/6,
     For[l = 0, l < 4, l = l + .25,
       Check[int = getIntersect[curve, boundaryList[[j]][[1]], l, k],
        Goto[skipped]]; (*If trying this intersect fails,
       skip checking whether it is the best intersect*)
       thisT = t /. int[[1]];
       thisTBoundary = t2 /. int[[2]];
       If[
        thisT > 0.00001 &&
         thisTBoundary > boundaryList[[j]][[2]] &&
         thisTBoundary < boundaryList[[j]][[3]],
        If[thisT < bestT ,
          Module[{},
           bestT = thisT;
           bestIndex = j;
           bestBoundary = boundaryList[[j]][[1]];
           bestInt = int;
           ]
          ];
        ];
       Label[skipped];
       ];
     ];
   ];
  Return[<|trajectory -> curve,
    trajectoryLength -> t /. bestInt[[1]],
    boundaryComponent -> bestBoundary,
    boundaryLocation -> t2 /. bestInt[[2]],
    boundaryIndex -> bestIndex,
    intersectionAngle ->
     FractionalPart[(getAngleFromCurve[bestBoundary,
           t2 /. bestInt[[2]]] -
          getAngleFromCurve[curve, t /. bestInt[[1]]])/(Pi) +
       5], (*This is a number between 0 and 1*)
    intersection -> bestInt|>]
  ]


(*Gets a number of consecutive trajectories from a starting \
trajectory on a given table.*)
guessTrajectoryListFinal[initial_, count_, boundaryList_] :=
 Module[{curve, int, plist, bestBoundary, intDetails, monitor},
  plist = List[];
  curve = initial;
  Do[
   monitor = PrintTemporary[i];
   intDetails = nextIntersectDetailed[curve, boundaryList];
   If[(intDetails[trajectoryLength]) < .00001,
    Module[{},
     Print["Possible error encountered: trajectory is very short"];
```

```
      Interrupt]];
    AppendTo[plist, intDetails];
    curve =
     getNextTrajectory[intDetails[trajectory],
      intDetails[boundaryComponent], intDetails[intersection]];
    NotebookDelete[monitor];
    , {i, 1, count}];
  Return[plist]]
```

```
getGeodesicCurvature[curve_] :=
 Module[{spaceCurve, curvature, unitNormal, ucurve, vcurve},
  ucurve = u /. curve;
  vcurve = v /. curve;
  spaceCurve[t_] := {f[vcurve[t]]*Cos[ucurve[t]],
    f[vcurve[t]]*Sin[ucurve[t]], g[vcurve[t]]};
  curvature =
   Function[t, Norm[Cross[spaceCurve'[t], spaceCurve''[t]]]];
  unitNormal =
   Function[t,
    Cross[Normalize[spaceCurve'[t]],
     Normalize[Cross[spaceCurve'[t], spaceCurve''[t]]]]];
  Print[curvature[1]];

  Return[Function[t,
    Cross[surfaceUnitNormal[(ucurve[t]), (vcurve[t])],
      Normalize[spaceCurve'[t]]].(curvature[t] unitNormal[t])]]]


(*Gaussian curvature of surface *)
K[v_] := (-g'[v]^2  f''[v] + g'[v] f'[v] g''[v])/(
   f[v]* (f'[v]^2 + g'[v]^2)^2)

(*The matrix T as a function of length t*)
getT[curve_] := Module[{v1, A, B, T},
  v1[t_] := v[t] /. curve;
  A := NDSolve[{J''[t] + (K[v[t]] /. curve)*J[t] == 0, J[0] == 1,
     J'[0] == 0}, {J}, {t, -1, 10}];
  B := NDSolve[{J''[t] + (K[v[t]] /. curve)*J[t] == 0, J[0] == 0,
     J'[0] == 1}, {J}, {t, -1, 10}];
  T[t_] := {{(J[t] /. A[[1]]),
     (J[t] /. B[[1]])},
    {(J'[t] /. A[[1]]),
     (J'[t] /. B[[1]])}};
  Return[T]]

(*Gets list of matrices to be multiplied together, in order \
...R_2,T_2,R_1,T_1*)
getSequenceOfMatrices[trajectoryList_] :=
 Module[{mList, curve, tcurve, boundary, tboundary},
  mList = List[];
  Do[
   curve = trajectoryList[[i]][[1]];
   tcurve = trajectoryList[[i]][[2]];
   boundary = trajectoryList[[i]][[3]];
   tboundary = trajectoryList[[i]][[4]];
   PrependTo[mList, getT[curve][tcurve]];
   PrependTo[
    mList, {{-1,
      0}, {2*getGeodesicCurvature[boundary][
         tboundary]/(Sin[
          getAngleFromCurve[boundary, tboundary] -
           getAngleFromCurve[curve, tcurve]]), -1}}];
   , {i, 1, Length[trajectoryList]}];
  Return[mList];]

TraceOfMatrices[lst_] := Tr[Dot @@ lst]


(*Returns 3D plot of surface*)
SurfacePlot[f_, g_, vmin_, vmax_] :=
 Module[{f1, f2, f3, F, Surface}, f1[u_, v_] := f[v] Cos[u];
  f2[u_, v_] := f[v] Sin[u];
  f3[u_, v_] := g[v];
  F[u_, v_] := {f1[u, v], f2[u, v], f3[u, v]};
  Surface := ParametricPlot3D[F[s, t], {s, 0, 2 Pi}, {t, vmin, vmax}];
  Return[Surface]]


EvalCoords[pt_] := {f[pt[[2]]] Cos[pt[[1]]], f[pt[[2]]] Sin[pt[[1]]],
  g[pt[[2]]]}

(*Gives trajectory (geodesic) starting on point on boundary*)
TrajectoryFromBoundary[boundary_, t_, angle_] :=
 geodesicSolve[u[t] /. boundary, v[t] /. boundary,
  angle + getAngleFromCurve[boundary, t]]
```

```
(*Gives Clairaut constant of curve. Assumes Clairaut function is \
indeed constant*)
ClairautConstant[curve_] :=
 Cos[getAngleFromCurve[curve, 0]] f[v[0] /. curve]

(*Gives Clairaut constant of curve. *)
ClairautConstant[curve_, t_] :=
 Cos[getAngleFromCurve[curve, t]] f[v[t] /. curve]


(*From trajectories, gives lists of points on the phase spaces of \
each boundary component*)
BilliardMapPhaseSpace[trajectories_, boundaryList_] := Module[{assoc},
   assoc =
    AssociationMap[Function[e, List[]],
     Map[Function[e, e[[1]]], boundaryList]];
   AppendTo[
      assoc[#[boundaryComponent]], {#[boundaryLocation], #[
        intersectionAngle]}] & /@ trajectories;
   Return[assoc];
   ];

(*Displays these phase spaces, one for each boundary component.*)
ShowPhaseSpace[assoc_, boundaryList_] :=
  Print[ListPlot[RemoveEmptyLists[assoc[#[[1]]]],
      PlotRange -> {{#[[2]], #[[3]]}, {0, 1}}]] & /@ boundaryList;


MakeAndShowPhaseSpace[trajectories_, boundaryList_] :=
  ShowPhaseSpace[BilliardMapPhaseSpace[trajectories, boundaryList],
   boundaryList];

(*Gets 3D plots of trajectories*)
TrajectoryPictures[trajectories_] :=
  getTrajectoryPlot[#[trajectory], #[trajectoryLength]] & /@
   trajectories;

(*3D plots of boundary components*)
BoundaryPictures[boundaryList_] :=
 getBoundaryPlot[#[[1]], #[[3]]] & /@ boundaryList

(*gets list of points aggregated from multiple orbits*)
PlotMultiplePhaseSpace[orbits_, boundaryList_] :=
  Module[{plotList, assoc},
   plotList = BilliardMapPhaseSpace[#, boundaryList] & /@ orbits;
   assoc =
    AssociationMap[Function[e, List[]],
     Map[Function[e, e[[1]]], boundaryList]];
   Function[f,
     Function[e, AppendTo[assoc[e[[1]]], f[e[[1]]]]] /@
      boundaryList] /@ plotList;
   Return[assoc];
   ];

(*Displays phase space from multiple orbits. Each orbit gets its own \
color of points*)
MakeAndPrintPhaseSpaceMultiple[orbits_, boundaryList_] :=
  ShowPhaseSpace[PlotMultiplePhaseSpace[orbits, boundaryList],
   boundaryList];

(*Since some orbits may fail to ever intersect a certain boundary \
component, this sanitizes lists to remove empty sublists so ListPlot \
will accept them.*)
RemoveEmptyLists[list_] :=
  Replace[list, x_List :> DeleteCases[x, {}], {0, Infinity}];



CompressOrbit[orbit_] := {orbit[[1]][trajectory],
   KeyDrop[orbit, {trajectory, boundaryComponent, intersectionAngle,
     intersection}]};

DecompressOrbit[compressed_, boundaries_] :=
 Module[{currentTraj, compressedOrbit, expanded},
  expanded = List[];
  currentTraj = compressed[[1]];
  compressedOrbit = compressed[[2]];
  Do[
   AppendTo[expanded,
    <|trajectory -> currentTraj,
     trajectoryLength -> compressedOrbit[[i]][trajectoryLength],
     boundaryComponent ->
      boundaries[[compressedOrbit[[i]][boundaryIndex]]][[1]],
     boundaryLocation -> compressedOrbit[[i]][boundaryLocation],
     boundaryIndex -> compressedOrbit[[i]][boundaryIndex],
     intersectionAngle ->
      FractionalPart[(getAngleFromCurve[
            boundaries[[compressedOrbit[[i]][boundaryIndex]]][[1]],
            compressedOrbit[[i]][boundaryLocation]] -
           getAngleFromCurve[currentTraj,
            compressedOrbit[[i]][trajectoryLength]])/(Pi) + 5],
     intersection -> {t -> compressedOrbit[[i]][trajectoryLength],
       t2 -> compressedOrbit[[i]][boundaryLocation]}|>];
   currentTraj =
    getNextTrajectory[currentTraj,
     boundaries[[compressedOrbit[[i]][boundaryIndex]]][[
      1]], {t -> compressedOrbit[[i]][trajectoryLength],
      t2 -> compressedOrbit[[i]][boundaryLocation]}]
   , {i, 1, Length[compressedOrbit]}];
  Return[expanded];
  ]

    (*This is a nice helper line that shows how long the cell takes to \
execute. It is great for determining whether improvements to the \
algorithm actually improve runtime.*)
SetOptions[$FrontEndSession,
 EvaluationCompletionAction -> "ShowTiming"]

(*The first thing we need to do is define our surface. *)
f[r_] := (1 - r^2)^(1)
g[r_] := r

(*Next, we need to define the boundary. All boundary components \
should be of the form {u\[Rule] \
__somePureFunction__,v\[Rule]__someOtherPureFunction__}. *)

(*The first built-in curve generator is for parallels. The function \
getParallel[v0,u0] returns a parallel at height v0 where t=0 \
corresponds to the angle u0.*)
bottom = getParallel[0, 0];

(*The next built-in curve generator (and the more interesting one) is \
for geodesics. The function geodesicSolve[u0,v0,\[Theta]] gives a \
geodesic on the surface where t=0 corresponds to the point (u0,v0) in \
the coordinate system of the surface, and the angle from a parallel \
at that point is \[Theta].*)
top = geodesicSolve[0, 0, Pi/4];

(*Next, we need to know how long each boundary component is going to \
be. In this example, we have only two boundary components, and we \
already they intersect when t=0 on both curves. To find the other \
point where they intersect (more importantly, the t-values of each \
curve where this intersection occurs), we use \
getIntersect[curve1,curve2,guessForCurve1,guessForCurve2]. It’s \
important to make sure the guess is pretty close, or else it might \
just throw us back the t=0 intersect, which, while correct, is \
unhelpful.It may be worth printing out the intersection just to make \
sure that the values look vaguely how we expected them to.*)
boundaryInt = getIntersect[bottom, top, 2, 2];

(*The return type of getIntersect is of the form \
{t\[Rule]__tCoordinateOfFirstCurve__,t2\[Rule]\
tCoordinateOfSecondCurve__} so to access these values one can use \
t/.int[[1]] and t2/.int[[2]] where int is the name of the \
intersection. *)


(*Before we can calculate any orbits, we need to explicitly tell \
Mathematica what defines our table's boundary. This is given by a \
list of triples. Each triple contains the bounday component (a \
curve), and the minimum and maximum t values that define the curve \
segment (real numbers). So in this example, even though the curve \
"bottom" really goes all the way around the parallel, its use as a \
boundary component is only between t=0 and the t value we found from \
the intersect. *)

tutorialBoundaries = {{bottom, 0, t /. boundaryInt[[1]]}, {top, 0,
    t2 /. boundaryInt[[2]]}};

    (*At long last, we are finally ready to calculate and view some \
orbits. We will start with one orbit. To construct an orbit, we need \
a starting trajectory. A useful way to create a starting trajectory \
is with TrajectoryFromBoundary[boundary,t,angle] which gives a \
geodesic (a trajectory) that starts on the boundary component at the \
given t value, and makes the given angle with that boundary (not with \
a parallel, as in geodesicSolve). In this example, we've created our \
starting trajectory (named billiard1) that starts on the bottom \
component, halfway between the two corners, and makes angle Pi/4 with \
the bottom component.*)

billiard1 :=
  TrajectoryFromBoundary[bottom, (t /. boundaryInt[[1]])/2, Pi/4];


(*To create a large number of trajectories on an orbit, we'll use \
guessTrajectoryListFinal[startingTrajectory,numberOfTrajectories,\
boundaryList]. The first intersect found will be the one after the \
starting trajectory intersects a boundary component, so the initial \
information of the starting trajectory will be lost. The program will \
calculate some number of trajectories as specified, and for this, it \
needs to have information about the boundary of the table (which is \
why we compiled it earlier.)

This program may take some time to run. It should print out a ticker \
with the number of trajectories calculated so far. It may be about \
5-10 seconds per trajectories, especially if your boundary has many \
components or your table is large, so please be patient. Many \
warnings may be printed out about various problems encountered, but \
so long as the ticker continues to increase, these can be safely \
ignored. They spawn from the program trying to find where the next \
intersection is going to be. If searching in the wrong area (so to \
speak), it may give up and throw an error. Luckily, since we're \
looking all over the place, we only need one of them to work out, and \
this usually does occur. *)
makeTrajectories =
  guessTrajectoryListFinal[billiard1, 35, tutorialBoundaries];



(*Welcome back! Thank you for waiting for your code to run. Now we \
can analyze our hard day's work. There are two primary things we can \
do with our data. First, we can draw pictures. When drawing pictures, \
we need to display three different things: trajectories, boundaries, \
and the surface itself. We can get the pictures of the trajectories \
and boundaries easily (using the trajectories we just calculated and \
the list of boundary information from earlier) and we can get a \
picture of the surface from SurfacePlot[f,g,vmin,vmax] which plots \
the surface between the given values of v. To show all of this, we \
wrap the surface plot in a one-element list,and join the three lists. \
Then Show will display the final work of art.*)

trajlist = TrajectoryPictures[makeTrajectories];
boundaryList = BoundaryPictures[tutorialBoundaries];
Show[Join[{SurfacePlot[f, g, -1, 1]}, trajlist, boundaryList]]


(*Next, we may want to see where the points on our boundary landed in \
the phase space. This method displays a number of 2D plots, one for \
each boundary component. Each has the x-axis as the t-value along the \
boundary component, and the y-axis as the angle from the boundary, \
scaled to (0,1). The true phase space would have these all plotted \
together side to side, but patterns may be easier to see if each \
boundary component gets its own graph.*)
MakeAndShowPhaseSpace[makeTrajectories, tutorialBoundaries];
```

```
(*We can also calculate many orbits at the same time to get a more \
complete picture of the phase space. To do this, we simply make a \
list, where each list item is itself a long list of trajectories. \
Table[listElement[i],{i,imin,imax}] makes this easier, where we can \
give the program parameters for what to calculate in each list \
element. In this example, we're fixing a point (in the middle of the \
bottom component) and we're calculating orbits that start with \
different angles from that point. The first orbit will start with \
angle Pi/30, the next with Pi/15, etc. This will take much longer \
than calculating the single orbit, but it can be sped up with \
Parallelize. Computations wrapped with Parallelize return the same \
value, but the utilize multiple cores on the machine. Since each \
orbit is independent of the others, we are free to use Parallelize \
without worrying about the calculation of one interfering with the \
calculation of another. This will print out a lot more errors, and a \
lot more progress numbers. Sorry. Still, as long as the numbers \
continue to go up, we're in good shape.*)

multipleOrbits =
  Parallelize[
   Table[guessTrajectoryListFinal[
     geodesicSolve[(t /. boundaryInt[[1]]), 0, Pi/30*i], 150,
     tutorialBoundaries], {i, 1, 14}]];


(*We can use the result of this to get nice pictures of the phase \
space, as shown below. Notice that any element of the list \
multipleOrbits has the same type as makeTrajectories above, so we can \
get 3D plots of any orbit by using \
TrajectoryPictures[multipleOrbits[[index]]].*)


MakeAndPrintPhaseSpaceMultiple[multipleOrbits, tutorialBoundaries]

(*From our list of trajectories, we can construct the list of \
matrices defining the derivative of the billiard flow. Matrices will \
be alternatingly matrices for a geodesic flow and a reflection. \
Importantly, we can get the trace of the matrix product easily.*)

trace = TraceOfMatrices[getSequenceOfMatrices[makeTrajectories]];

(*Suppose we let our computer do the hard work of calculating a large \
number of trajectories, but we wish to shut our machine down for the \
weekend without losing all this valuable information. Luckily, we can \
save the information in a file and recall it later. Unfortunately, \
saving a whole orbit with all its information is too much information \
for the computer to handle, as each interpolating function is roughly \
1Mb of information. We opt to compress our data substantially, and it \
will be quick to recover later. After this, we store the data in a \
file for later use. We need to pick a location to put our new file, \
and the location of the notebook is a suitable choice, so we change \
the directory to the notebook's directory. ".m" is the typical choice \
of filetype for such files. The new file simply contains the text of \
the value we stored, as if we had printed it to the screen.*)

compressedSingleOrbit = CompressOrbit[makeTrajectories];
SetDirectory[NotebookDirectory[]];
compressedSingleOrbit >> "tutorialSingleOrbit.m"

(*Now we're back and ready to pull up our information from earlier. \
We can pull it out of the file much the same way we put it in. \
Decompressing is also straightforward, we only need the boundary \
list. Reinitializing the boundary list should be quick.*)

compressedSingleOrbit = << "tutorialSingleOrbit.m"
makeTrajectories =
  DecompressOrbit[compressedSingleOrbit, tutorialBoundaries];

(*We are now back to where we were, and we can proceed with data \
manipulation.*)

(*It is also easy to compress and decompress multiple orbits using \
Map.*)
compressedMultipleOrbits = CompressOrbit /@ multipleOrbits;
SetDirectory[NotebookDirectory[]];
compressedMultipleOrbits >> "tutorialMultipleOrbits.m";

compressedMultipleOrbits = << "tutorialMultipleOrbits.m";
multipleOrbits =
  DecompressOrbit[#, tutorialBoundaries] & /@
   compressedMultipleOrbits;
```